Small or medium-scale focused research project (STREP)

ALMA

Architecture oriented paraLlelization for high performance embedded Multicore systems using scilAb

FP7-ICT-2011-7
Project Number: 287733

Deliverable D3.8

ALMA design framework guidelines for multicore systems (final version)

Editors:	George Goulas/TMES
Authors:	George Goulas, Panayiotis Alefragis, Nikolaos Voros Oliver Oey, Thomas Bruckschloegl, Timo Stripf Ali Hassan El-Moussawi, Mythri Alle

Status – Version:	1.02
Date:	10/02/2015
Confidentiality Level:	PUBLIC
ID number:	FP7-ICT-2011-7-287733-WP3-D3.8-v1.02.doc

© Copyright by the ALMA Consortium

The ALMA Consortium consists of:

Karlsruher Institut fuer Technologie	Coordinator	Germany
Universite de Rennes I	Contractor	France
Recore Systems B.V.	Contractor	Netherlands
University of Peloponnese	Contractor	Greece
Technologiko Ekpaideftiko Idryma Mesologgiou (Technological Education Institute of Messolongi)	Contractor	Greece
Intracom SA Telecom Solutions	Contractor	Greece
Fraunhofer-Gesellschaft zur Foerderung der Angewandten Forschung E.V.	Contractor	Germany

Document revision history

Date	Version	Editor/Contributor	Comments
22/12/2013	1.00	George Goulas/TMES	Final Version
06/02/2015	1.01	Thomas Bruckschlögl/KIT	Typos, Reference update
10/02/2015	1.02	Thomas Bruckschlögl/KIT	Fixed Table of Contents

Preface

This document is a collection of end-user guidelines for the ALMA tool chain. The user perspective for most parts of the ALMA tool flow components will be presented. It provides advice for an optimal use of the ALMA tool chain to create faster applications on Multiprocessor and Multicore systems.

Abstract

The ALMA project main product is the ALMA tool chain that is an end-to-end tool chain from a high-level program representation directly to embedded Multiprocessor System-on-Chip (MPSoC) platforms. The ALMA tool chain is not tailored to a specific platform, but rather abstracts the underlying parallel hardware architecture by the introduction of the Architecture Description Language. Various intermediate code representation optimizations, including polyhedral analysis and control and data flow graph optimization are introduced, before the parallel code is presented to the platform specific tools. This document provides end-user guidelines towards extracting the best performance from applications created using the ALMA tool chain.

List of Figures

Figure 1: ALMA tool flow from the user perspective 8
Figure 2: ALMA tool chain architecture........ 9
Figure 3: Fine-grain Flow 15
Figure 4: Multicore Simulator........ 23
Figure 5: Parallel Matrix Multiplication Scilab Code – File `matrixmul_parallel.sce` 28
Figure 6: GeCoS Script `matrixmul_parallel.cs`: First part, Front End 28
Figure 7: GeCoS Script: Coarse grain parallelism extraction and fine grain parallelism optimization 29
Figure 8: GeCoS Script demonstrating the use of CoarseGrain 30
Figure 9: The top level HTG layer for `matrixmul_parallel` 31
Figure 10: The parallel processor assignment for the top level HTG layer for `matrixmul_parallel` (processors are encoded as colours)........ 31
Figure 11: GeCoS Script `matrixmul_parallel.cs`: parallel code generation........ 32
Figure 12: GeCoS Script `matrixmul_parallel.cs`: parallel code generation........ 32
Figure 13: GeCoS Script: compiling and simulating the application........ 33

List of Tables

Table 8-1: Command line argument list for systemsim 23
Table 8-2: Command line argument format 24
Table 8-3: Available systemsim commands........ 24
Table 8-4: Descriptor for module-specific command line arguments........ 24
Table 8-5: Escape character for module-specific descriptor 25
Table 8-6: Descriptor for instance-specific command line arguments........ 25
Table 8-7: Escape character for instance-specific descriptor........ 25
Table 8-8: Configuration file format 26
Table 8-9: Kahrisma specific simulator options........ 27
Table 8-10: Extract of Xentium simulator options. 27
Table 8-11: Extract of Xentium profiler options. 27

Table of contents

1 5

1 INTRODUCTION 7

2 THE ALMA FRAMEWORK 7

3 DESIGN METHODOLOGY 10

4 ALMA FLOW FRONTEND TOOLS 11

4.1 ALMA PRAGMA SUPPORT 11
4.1.1 alma_task_cluster 11
4.1.2 alma_task_alloc 11
4.1.3 gcs_scop_ignore 12
4.1.4 gcs_scop_cg_schedule 12
4.1.5 GCS_PURE_FUNCTION 13
4.2 BEST PRACTICE FOR WRITING INTERFACE CODE 13

5 FINE GRAINED OPTIMIZATIONS FLOW 15

5.1 OVERVIEW 15
5.2 LOOP TRANSFORMATIONS 15
5.2.1 Specifying Register Level Tile sizes 16
5.2.2 Understanding SCoPs 16
5.3 FLOATING-POINT TO FIXED-POINT CONVERSION 18
5.3.1 Applicability of Automatic fixed-point specification 18
5.3.2 Specifying Accuracy Constraint 18
5.4 SIMD VECTORIZATION 19
5.5 GENERAL SCILAB CODING GUIDELINES 19

6 COARSE GRAIN PARALLELISM EXTRACTION 19

6.1 OVERVIEW 19
6.2 PARAMETERS CONTROLLED BY USER 20
6.2.1 Number of parallel tasks 20
6.2.2 Tile size 20

7 COARSE GRAIN PARALLELISM OPTIMIZATION 20

7.1 OPTIMIZATION METHOD OPTIONS AND TRADE-OFFS 22

8 MULTICORE SIMULATOR 22

8.1 COMMAND LINE INTERFACE 23
8.1.1 Global Command Line Arguments 23
8.1.2 Module Specific Command Line Arguments 24
8.1.3 Instance-Specific Command Line Arguments 25
8.1.4 Providing Application files 25
8.2 CONFIGURATION FILES 25
8.2.1 Kahrisma Specific Options 26
8.2.2 Xentium Specific Options 27

9 ALMA FLOW DEMONSTRATION 27

9.1 FRONT END – GENERATING C CODE 28
9.2 COARSE AND FINE GRAIN PARALLELISM EXTRACTION 28
9.3 COARSE GRAIN PARALLELISM OPTIMIZATION 29
9.4 PARALLEL CODE GENERATION 32
9.5 SIMULATING THE GENERATED PROGRAM 32

10 CONCLUSIONS 33
11 REFERENCES 33

Glossary of Terms

CAS	Cycle Accurate Simulator
CDFG	Control and Data Flow Graph
CLI	Command Line Interface
DDL	Data Description Language
ELF	Executable and Linking Format
GeCoS	Generic Compiler Suite
ILP	Instruction Level Pipelining
IR	Intermediate Representation
ISS	Instruction Set Simulator
KAHRISMA	KArlsruhe's Hypermorphic Reconfigurable-Instruction-Set Multi-grained-Array Processor
LTI	Linear Time-Invariant
MPSoC	Multiprocessor System on Chip
SCoP	Static Control Part
SIMD	Single Instruction Multiple Data
SoC	System-on-Chip
SSA	Static Single Assignment
SWP	Sub-Word Parallelism

1 Introduction

This document presents guidelines specific to the ALMA methodology and how the ALMA tool chain can be effectively used for the design and development of applications for embedded computing systems relying on multicore architectures. The ALMA tool chain receives a special Scilab-based [1] dialect and produces binaries for the designated Multicore Parallel System-on-Chip (MPSoC) platform.

The following section presents a high-level overview of the whole ALMA framework; more information about the ALMA framework can be obtained in [3,4,5,6]. A design methodology specific for the ALMA tool chain follows, which summarizes the guidelines presented in this document. Front-end tools, the input language and intermediate code optimizations are presented next. Guidelines for the two big optimization engines, the fine grain parallelism engine and the coarse grain parallelism engine, are presented in the following sections. These sections give a user-level overview for the engines as well as advice for a better exploitation of those engines. The ALMA multicore simulator is presented afterwards. The simulator participates in an internal optimization loop to improve the results of the optimization engine and is also available to the end users in order to benchmark and profile their applications. A demonstrator on the use of the ALMA components follows. In the document annex, the *ALMA Toolchain User Guide* and the *Matrix Frontend User Guide* are provided for reference.

2 The ALMA Framework

The ALMA framework offers a platform agnostic tool flow from Scilab [1] input code directly to Multiprocessor System-on-Chip (MPSoC) architectures. The Scilab language subset supported by the ALMA tools is described in D4.2 [7]. It also provides a detailed technical description of the tool chain.

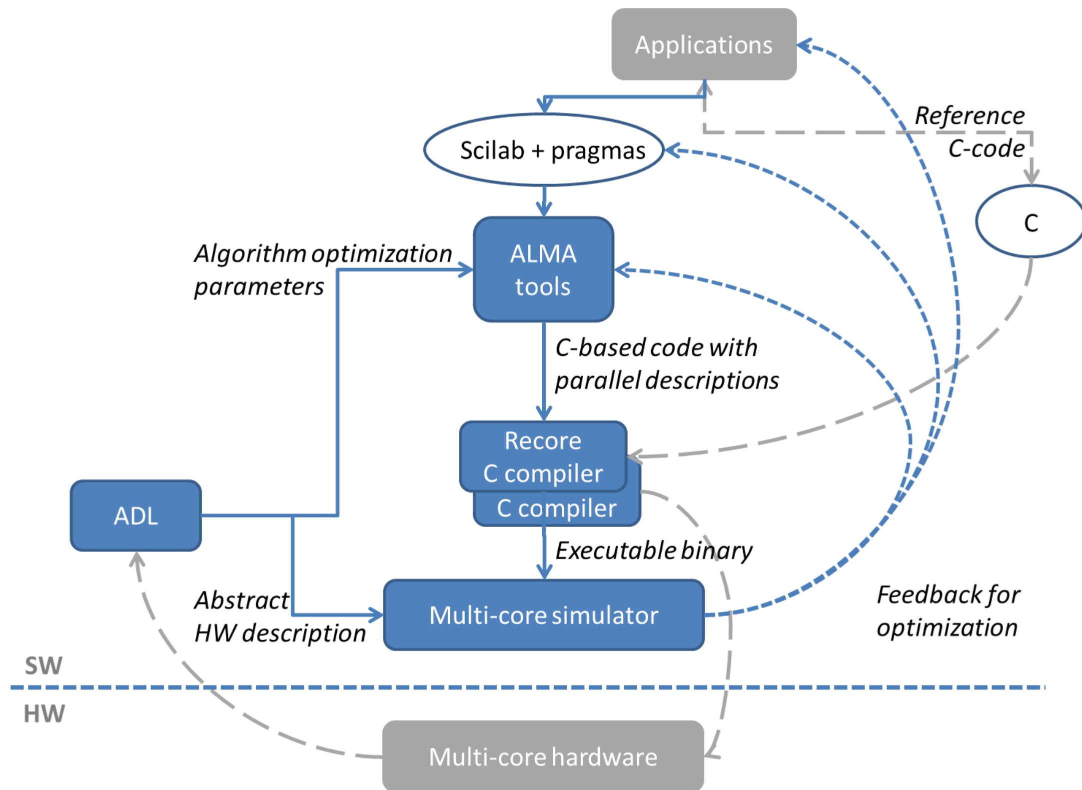

Figure 1: ALMA tool flow from the user perspective

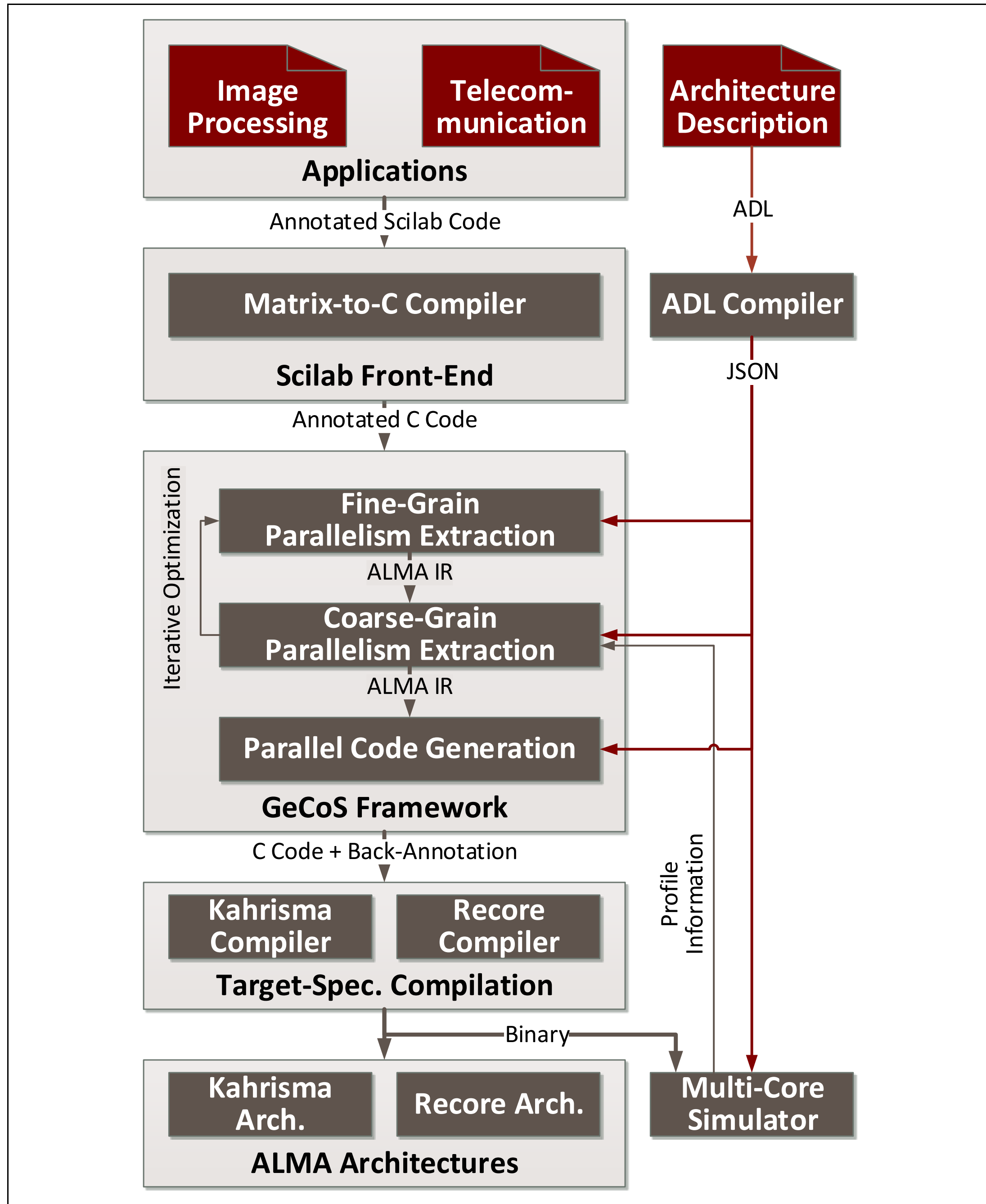

Figure 2: ALMA tool chain architecture

A user perspective of the ALMA tool flow is presented in Figure 1. The components of the ALMA tool chain are presented in Figure 2. The inputs to the ALMA tool flow are the program code in Scilab and the architecture definition given in the Architecture Description Language (ADL) format. The primary output of the tool flow is a set of binaries for the parallel application, ready to run optimised on the MPSoC described in the ADL input. Front-end tools initially process the Scilab input in order to convert it in a convenient C Intermediate Representation (IR) and perform static code analysis and optimization. Following this step, two different optimization approaches extract available parallelism from the application. The two optimization approaches are implemented within the GeCoS Generic Compiler Suite [2]. The fine

grain optimization focuses on nested loops and performs polyhedral analysis type optimizations, while the coarse grain optimization considers the application control and data flow graphs and performs graph-based optimizations. Based on the result of the above optimizations, a parallel version of the intermediate code representation is generated. This code is provided to the platform specific tool-chain, where further platform specific optimizations occur. The resulting application is profiled and benchmarked by the platform simulator. The results are used in a feedback loop with the coarse grain parallelization, which creates an improved version of the program. This feedback cycle can be repeated until the user considers the results satisfactory in terms of performance.

3 Design methodology

Although the ALMA tool flow does not expose the user to the parallel programing challenges and requires only sequential programming, it is useful to present an overview for the typical parallel programming workflow in order to achieve application performance.

The first step towards a parallel application usually involves the creation of an equivalent sequential application. This step is not redundant, since it allows the developers to understand the problem at hand, identify potential bottlenecks and opportunities for parallelization and most important, establish reference results in order to validate the final application components. These sequential application prototypes are usually implemented in a high level language like Scilab or Matlab, while the parallel applications are typically implemented in C/C++.

When a sequential reference application is implemented, profiling and benchmarking are crucial to identify real performance bottlenecks. If the sequential application is implemented in the same language as the parallel one, the same code base can be reused and a set of experiments with different parallelization strategies can be performed. This code evolution strategy has the advantage, that a correct application exists, but it should be noted that it is not always feasible, as parallel algorithms might require substantially different data structures and approaches. The set of alternative parallelization strategies to different sections of the application are evaluated for performance through profiling and benchmarking. The developers will decide, the appropriate alternative strategies that are going to be part of the final application. The development process finishes when the application satisfies operational requirements.

In the ALMA tool flow, the user provides a high level sequential implementation and the ALMA tool chain is responsible to perform the necessary transformations in order to generate an equivalent parallel version that uses the underlying multicore architecture. As mapping and scheduling decisions are made at compile time, user supplied input should be as representative as possible, as the tool chain uses this to perform optimizations that are based on feedback from performance measurements. As a result, parallel programs that exhibit high performance fluctuations to different inputs can be heavily affected, if real world input examples are not provided to the ALMA toolchain by the end user. A large set of supplied inputs will lead to slow compilation times for the tool chain, but will steer the generated parallel code to more robust performance. The end user is advised to balance between fast exploratory builds with a limited number of input data sets and slow release builds with extended number of input datasets. In addition, the user can perform additional application benchmarking and profiling with direct use of the platform simulator.

Although the goal of the tool chain is to free the application designer from parallelization details, specific source code features can result in substantial performance benefits, as it is analysed in this document. Specifying accuracy constraints on variables enables the optimizing tools to exploit sub-word parallelism (SWP). Specifying loop nests as Static Control Parts (SCoP) enable fine grain parallelism optimizations. Defining reusable code fragments as functions helps the coarse grain parallelism tool to treat them as candidate independent tasks. As mentioned before, providing representative program inputs enhances the available information for the coarse grain simulation-based optimization and thus higher quality parallel programs are generated. Balancing computation workload between alternative paths of execution in the sequential code also allows for more predictable execution and better use of the available resources of the underlying architecture. Code sections, for which the coarse grain optimization has problems to predict execution times, like recursive functions or loops whose limits depend on the input, can pose a significant challenge for the coarse grain optimization module, which relies on simulator profiling. Defining variables as scalars instead of vectors can increase performance.

4 ALMA flow frontend tools

The Matrix Frontend (MFE) is a source-to-source translation utility that converts the Scilab language into static C code. The documentation of the MFE is available within the Matrix Frontend User Guide. A copy of this document is available in the Appendix of this report.

The frontend comprises a generic pragma interface. The interface is used by the ALMA Toolchain to define ALMA-specific pragma to drive the parallelization approach. In the following section, this interface is specified for the end users in addition to the MFE documentation (provided in the Annex to this document).

4.1 ALMA pragma support

The frontend supports three function-like pragmas, that will generate #pragma instructions within the C code. A pragma can be either attached to an instruction (mfe_pragma), a block (mfe_pragma + if), a variable (mfe_pragma_var) or a function (mfe_pragma_func). It depends on the type of pragma, what methods should be used.

In the following, a list of ALMA-specific pragmas and their usage in the front end is given:

4.1.1 alma_task_cluster

This pragma enforces the coarse-grain scheduling to assign the annotated block to one processor only.

Usage:

```
//MFE?: mfe_pragma('alma_task_cluster');
if (1)
      …
end
```

4.1.2 alma_task_alloc

This pragma manually assigns the annotated block to a given processor. The processor is give as a processor id.

Usage:

```
//MFE?: mfe_pragma('alma_task_alloc <id>');
```

```
if (1)
      …
end
```

4.1.3 gcs_scop_ignore

The part of the code is ignored for polyhedral optimization within the fine- and coarse-grain optimization stage of the ALMA tool flow. In general, polyhedral optimization offers some performance benefits coming with the price of increased C code complexity and size. The pragma allows skipping this optimization for parts of an application that is not relevant for optimization.

Usage:

```
//MFE?: mfe_pragma('gcs_scop_ignore');
if (1)
      …
end
```

Deactivates polyhedral optimizations within the if-block.

```
function […] = myfunc(…)
      //MFE?: mfe_pragma_func('gcs_scop_ignore');

      …
end
```

Deactivates polyhedral optimizations within the function.

4.1.4 gcs_scop_cg_schedule

This pragma allows to specific parameters for the polyhedral coarse-grain optimization. The parameters are valid for the given block if it is amenable to polyhedral analysis.

Usage:

```
//MFE?: mfe_pragma('gcs_scop_cg_schedule [numbertasks <int>]
[fixedtasks <id*>] [tileSizes <int*>]');
if (1)
      …
end
```

numbertasks specifies the number of concurrent tasks that will be extracted during the coarse-grain pass. If not specified, the global value is used as defined in the compilation script (cs).
fixedtasks specifies a concrete allocation of tasks to processors. The processors are given as ids in a comma separated list. That is independent from the number of tasks extracted. For tasks a processor ID is provided, the concrete allocation is performed by alma_task_alloc, otherwise alma_task_clusters is used.

tileSizes specifies the tile size parameter for each loop in a loop nest. The first parameter is used for the outermost loop. The second for the loops within the outermost loop, etc. If there exists more nested loops than parameters, the last parameter is used for the remaining loops without parameter.See section 5.2 for more information about the effect of tile size.

4.1.5 GCS_PURE_FUNCTION

This pragma allows the specification of pure functions. Pure functions are deterministic functions without side effects i.e. that do not modify global variables. It is allowed to call the function simultaneously from different threads without breaking the semantic. Function calls to pure function can be analysed using polyhedral analysis.

Usage:

```
function […] = myfunc(…)
      //MFE?: mfe_pragma_func('GCS_PURE_FUNCTION');


      …
end
```

4.2 Best practice for writing interface code

The developed applications are used for two cases:

1. For the simulator to generate profiling results that drive the parallelization process.
2. For the hardware to generate the product code.

For both cases, the code differs only in the interface code. For the simulator, the interface code includes loading the test data and is directly specified within the frontend. For the hardware, the interface code must be replaced by hardware-specific code.

A real-time application typically runs in an infinite loop. The basic idea is to express the infinite loop within Scilab and include function calls to the interface code. Within the frontend, also an implementation of the interface function is available that is used for profiling the application within the simulator as well as defines the interface of the function in C code. The implementation of the interface function is generated into a separate C file. For the hardware, the C file can be replaced by a custom implementation that accesses the hardware interfaces to get the data in and out.

In the following example, a best practice skeleton in combination with data in/out functions is provided:

```
function runs = interface_numruns()
      //MFE?: mfe_func_noinline();
      //MFE?: mfe_func_file('interface.c');

      runs = 2;
end


function [A,B] = interface_indata(run, N)
```

```
	//MFE?: mfe_func_noinline();
	//MFE?: mfe_func_file('interface.c');

	A = int32(eye(N,N));
	A(1,2) = -3;
	A(2,1) = run;

	B = int32(matrix(1:N^2,N,N));
end

function interface_outdata(run, C)
	//MFE?: mfe_func_noinline();
	//MFE?: mfe_func_file('interface.c');

	disp(C);
end



N = 4;
runs = interface_numruns();

//MFE?: mfe_pragma('alma_infinite_loop');
for run = 1:runs
	//MFE?: mfe_pragma('alma_task_alloc 0');
	if (1)
		[A,B] = interface_indata(run, N);
	end

	C = A*B;

	//MFE?: mfe_pragma('alma_task_alloc 0');
	if (1)
		interface_outdata(run, C);
	end
end
```

The interface code consists of three functions:

interface_numruns returns the number of runs the “infinite” loop should run. For simulation, that is set to 2 in the example. In the manual implementation of the interface code for hardware, the function can return a very high number to keep the application running in the infinite loop.
interface_indata reads data. In this example, two matrices are generated for simulation and profiling.
interface_outdata writes data. In this example, the resulting matrix is displayed for simulation.

The code uses several pragmas and attributes. For each interface function, the mfe_func_noinline is specified that prevents constant values from being propagated out of the function. This is important since the functions are completely replaced in the product code. Also for each interface function, the output C file is specified. The generated interface.c file can be used for simulation, for the product code a separate file can be implemented.
The main loop is tagged as an infinite loop, to hit the coarse-grain extraction to use task-level pipelining.
The calls to the interface functions are pinned to processor 0 that should perform the IO.

5 Fine grained Optimizations Flow

Obtaining an efficient implementation requires not only to efficiently distribute the original program execution on several processors, but also to ensure that the code executed on each core is highly optimized with respect to its target architecture.

In this section, we present an overview of the fine-grain optimization flow, integrated in the ALMA tool chain. We then provide some user guidelines to efficiently exploit this flow. The reader is invited to refer to ALMA deliverable D3.6 [10] for technical details.

5.1 Overview

The fine-grain optimizations flow provides a transformation toolbox including:

- Several loop transformations to enhance data locality and reuse
- Automatic SIMD (Single Instruction Multiple Data) vectorization to leverage SWP (Sub-Word Parallelism) capabilities of target architectures
- Floating-point to Fixed-point conversion.

Figure 3 presents an overview of the fine-grain optimization flow. Guidelines about these optimizations are presented in the following sections.

5.2 Loop Transformations

Register Level Tiling is applied in order to enhance data locality. This transformation is done within the polyhedral framework, so it is only applicable on Static Control Parts (SCoP) of the application. This transformation also annotates full tile basic blocks (kernels) within loop bodies. Only these basic blocks will be considered for later optimization, since they usually represent the largest part of the loop nest computations. For non-polyhedral parts, basic blocks can be manually annotated using pragma "__RLT_INSET__".

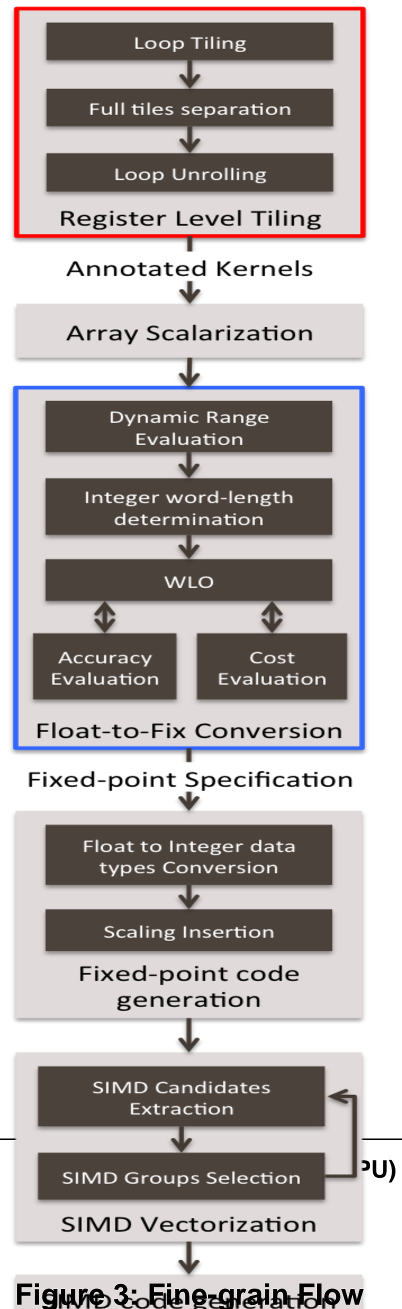

Figure 3: Fine-grain Flow

The next optimization, Array Scalarization, exposes data reuse. It can be applied on any basic block, though in the fine-grain flow it is only applied on annotated kernels.

5.2.1 Specifying Register Level Tile sizes

Register Level Tile sizes can be specified either using

- a global parameter specified in the GeCoS script
- or via pragma annotations in the Scilab code using pragma "gscop_fg_schedule tileSizes=[sizes…]".
- If no tile size is given then the default size is used. Default tile size is 2

When choosing a tile size, the user must keep in mind that this will determine the size of the generated kernel (after point loop unrolling). This consequently determines the number of required registers to do the computation, it also control the amount of exposed SWP. Therefore, tile size should be big enough so that enough SWP is exposed, yet small enough to avoid register spilling. Intuitively, bigger tile sizes yield larger kernels (basic block), which contain more computations, consequently more SWP opportunities are potentially exposed but more registers may be required. Detailed information about Register Level Tiling is available in ALMA deliverable D3.6 section 6.

5.2.2 Understanding SCoPs

Our loop transformations can only operate on a subset of programs known as Static Control Part (SCoP). The end user must hence understand what SCoP are and how to make sure their scilab programs/kernels fall in this category.

SCoPs consist of perfectly or imperfectly nested for-loops with statements that access arrays or scalar variables. In a SCoP both loop bounds and arrays indexing functions must be affine (i.e. linear) expressions[1] of *dimension indices* and on some *parameters*. The *dimension indices* correspond to all the loop indices enclosing a given statement. Parameters are variables that may not be known at compile time, but which remain constant during the SCoP execution (the size of an array is a good example of such parameters).

For example the expression 2*j may be affine if j is an enclosing loop index (or, more unlikely, a parameter). However N*j is not affine as neither j nor N are compile-time constants.

Below, we provide four examples illustrating the previous definition.

```
for i=1:10
  x(P+i-1)=i;
  for(j=1:P)
     if(j+i<10)
        z(j)=x(i+j)+x(j);
     end
  end
end
```

```
for i=1:10
  x(size_x-1)=i;
  for j=1:Z
    z(j) = x(j) + x(j);
  end
  Z = … ;
end
```

[1] The notion of affine expression and/or constraints needs to be understood in a broader way than usual, as it also comprises quasi-affine expressions (involving integer division by a constant). Such quasi-affine expressions are simply rewritten using affine expressions (usually with one or more extra variable).

```
for i=1:N
  x(size_x-1)=i;
  for j=1:10
    z(j*N+i)=x(j)+x(j);
  end
end
```

(a) The example above is a SCoP, as it only consists of affine array indexing function and affine loop bound.

(c) This loop nest is not a SCoP, as array z[] is indexed using a non affine function.

```
for i=1:N
  x(size_x-1)=i;
  if x(i)>0
    for j=1:10
      z(j,i)=x(j)+x(i);
    end
  end
end
```

(b) This loop nest is not a SCoP, as variable Z is modified in the outer loop and is used as a bound in the inner loop.

(d) This loop nest is not a SCoP, as the innermost loop execution is guarded by a data-dependant condition.

The SCoP identification and extraction stage in ALMA is performed using a combination of syntactic pattern matching and program transformations. It is able to detect most SCoPs in a program.

However, there are situations where a program subset is not a SCoP (syntactically speaking) but *behaves* like a SCoP. For example, it is obvious for a programmer, that in the example (e), the recursive expression used for indexing array A is an affine expression of the loop index. Inferring this information (in the general case) from source code is challenging and is not supported in the flow. As a general guideline, array access expression should always be explicit affine expression of loop indices and parameters as in example (f).

```
tmp=0;
for i=1:10
   tmp = tmp +3
   A = A (tmp);
end
```

(e) Implicit affine array index expression

```
for i=1:10
   A = A (3*i);
end
```

(f) Explicit affine array index expression

More generally, it is often the case, that the SCoP detection is hindered by only one or two statements in a loop nest, even if the loop contains several tens of such statements. Similarly, some kernels do expose a small/local data dependant behaviour, which would not prevent the loop to be (at least conceptually) modelled as a SCoP, but cause the program to be flagged as non SCoP.

A simple solution to this problem is to perform minor program modifications to hide complex behaviour within side-effect free external functions as shown in the examples (g) and (h).

```
for i in 1:10
   if A(i)*X(i)>1.0
     sum = sum +1;
     for j in 1:10
       A(j) = A(j)*X(i)
     end
```

```
function [res,newA] update(sum,i,A,X)
   newA=A; res = sum;
   if A(i)*X(i)>1.0
     res = sum +1;
     for j in 1:10
       newA(j) = newA(j)*X(i)
```

```
   end;
end;
```

(g) This loop nest is not a SCoP kernel since there is a data-dependant guard enclosing the innermost loop.

```
      end;
   end;
end
...

for i in 1:10
   [sum, A] =update(sum,i, A,X);
end;
```

(h) This loop is now a SCoP, we have hidden the data-dependant behaviour within a function. Note however the resulting SCoP is a single loop (the j loop is not part of the SCoP, as it is hidden in the function)

5.3 Floating-point to Fixed-point Conversion

In a first step, this transformation aims at obtaining a valid fixed-point specification for each floating-point data/operator in the original program. Then, it generates the corresponding fixed-point c code using native c integer types and shift operations to properly perform the required scaling. The first step can be performed either:

- automatically (blue section in Figure 3), only supported for a sub-class of programs as described below
- manually in the Scilab/Matlab code using fixed-point types which can be created by Matlab-like function *fi* (see Matrix Frontend User Guide).

5.3.1 Applicability of Automatic fixed-point specification

An automatic fixed-point specification can be obtained for programs that satisfy the following criteria:

- All loop bounds and guard conditions must be known at compile time
- Accessing data using pointers is not supported
- The underlying system must be LTI (Linear Time-Invariant)
- All system input variables must be annotated with their dynamic range using pragma annotation “DYNAMIC [min, max]”
- The system output variable must be annotated with pragma “OUTPUT”
- System delay variables can be specified using pragma “DELAY”

When these criteria are respected the tool can automatically explore the fixed-point design space using WLO (Word-Length Optimization) algorithms. It tries to minimize the implementation cost (execution time) subject to an accuracy constraint. In this case the user must provide an accuracy constraint, that is the maximum accepted SQNR (Signal to Quantization Noise Power), measured in dB,at the system output.

5.3.2 Specifying Accuracy Constraint

The accuracy constraint can be specified in the GeCoS script as a string. It represents the maximum noise power, given in dB, allowed at the system’s output. The floating-point to fixed-point conversion process makes sure that the noise power of the selected fixed-point solution is less than or equal the accuracy constraint value. Smaller values represent a stricter accuracy constraint.

5.4 SIMD Vectorization

Although, SIMD vectorization can be applied on any basic block, we only apply it on those blocks annotated with pragma "__RLT_INSET__". This annotation is set automatically for SCoPs and can also be set manually in Scilab code as described previously.
A higher vectorization factor can be achieved, when the data word-length is narrower. So the user is advised to use smallest possible data types.

5.5 General Scilab Coding Guidelines

The coding style used in the Scilab program specification may influence the efficiency of the fine-grain optimization stages. In the following, we provide simple guidelines to help end user to write adequate Scilab programs.

Whenever possible, the end user should use constant (static) size matrices /vectors in the program. When this is not possible, the number of statements, which alter the shape of an object, should be minimized (ideally the object size is defined in an initialisation statement which is visible to all program execution paths). For example, for the two examples below that perform the same operation, the formulation (b) is likely to hinder the SWP kernel extraction stage.

```
A = zero (10,10);
for i in 1:10
   A(i) = x(i)
end;
```

(a) Runtime constant array size

```
for i in 1:10
   A = [ x(i) A ]
end;
```

(b) Runtime dependant array size

6 Coarse grain parallelism extraction

This section provides a high level overview of the coarse grain parallelism extraction pass and describes parameters that can be controlled by the user to fine tune the performance of the applications.

6.1 Overview

This pass extracts parallelism, that is available in loops using loop tiling transformations. Loop tiling was initially proposed to improve locality in loops. Loop tiling changes the order of execution to enable re-use of data. This transformation basically partitions the iteration space into several blocks, called tiles, and these blocks are executed one after another. One important property of these tiles is that they can be considered as "atomic" entities. Hence, these tiles can receive all the data required by the tile at the beginning of the execution and send the data once the execution is complete. In other words, the tiles can be treated as a coarse-grained communication entity.

Another interesting property of these tiles is, that they exhibit wave-front parallelism i.e. all the tiles in the anti-diagonal can be executed in parallel. This property is used to extract the required amount of parallelism. The tiles are distributed in a block-cyclic manner to the required number of parallel tasks. More technical details regarding this transformation can be found in ALMA Deliverable D3.5 [9].

6.2 Parameters controlled by user

Tile size and the number of parallel tasks are the two important parameters, that can be controlled to tune the performance of the applications. In the following section, we present few guidelines to choose these two parameters.

6.2.1 Number of parallel tasks

One of the guiding factors for choosing the number of parallel tasks could be based on the number of processors available on the hardware. The number of processors available places an upper limit on the number of parallel tasks that have to be extracted. Other important factors include the latency of communication and the problem size.

6.2.2 Tile size

Choosing the right tile size is an important factor, that determines the efficiency of this optimization. Since tiles are treated as a communication entity, the size of the tile determines the amount of data that needs to be communicated. In other words, the size of the tile determines the compute to communication ratio. Thus, the size of the tile is influenced by the communication latency and also the complexity of computation inside a loop. Also, to enable the reuse of data within a tile, the cache should be able to hold the data required by each tile. Thus, cache size is another factor that influences the choice of the tile size. Hence, choosing the right tile size is a difficult task.

However, as a general rule, small tile sizes (less than 16) are not efficient, since the communication has to be performed very frequently, which affects the overall performance. The tile size also determines the maximum number of parallel tasks that can be extracted. Thus, the maximum tile size can be computed based on the number of parallel tasks required. If the number of parallel tasks required is n and the number of iterations in a particular dimension is S, thenthe tile size should be less than S/n. For example, for a matrix multiplication example with 512 X 512 array size, if we need 4 processors, the maximum tile size can be 128 X 128. However, it is preferable, that the tile size is not more than 64 X 64 to have more parallelism.

The number of parallel tasks and the tile size are set to a default value of 4 and 16 respectively. These values can be controlled by passing them as parameters when invoking this pass. They can be also controlled from the Scilab application by specifying pragmas. These pragmas give the user more control over these parameters. Using pragmas, these parameters can be set to different values for different portions of the code.

7 Coarse grain parallelism optimization

During the coarse grain parallelism extraction and optimization (CGPEO) phase, the program Control and Data Flow Graph (CDFG) is modelled and optimized for minimum total execution time. This section attempts to provide an user overview of the whole phase, so that the user will be able to assist the CGPEO tools. CGPEO tools receive C source code that is produced by the MFE.

The CDFG representation of a program includes sets of program instructions that have single control input and output connections, named basic blocks. A pre-processing step identifies procedure calls and splits basic blocks accordingly, in order for procedure calls to be identified as single blocks. As a result, procedure calls

define starting points for the CGPEO process. Another pre-processing step splits basic blocks, where instructions do not have data dependencies between them and, as a result, can be executed in parallel.

Following the pre-processing steps, the CDFG is converted to a Hierarchical Task Graph (HTG). In the HTG form, the blocks that have occurred after the pre-processing steps form an interconnected set of Directed Acyclic Graphs (DAG). In order to remove cycles from the CDFG, single blocks are considered as leaf nodes. All other nodes that contain complex structures define a complex node, and contain a new hierarchical layer in the HTG, which itself is a DAG that contains leaf or complex nodes. For every DAG that has nodes that refer to a loop body, a procedure call or an if/else block, a new lower hierarchical layer is defined and the corresponding DAG that represent the aforementioned constructs are moved to the new level. The process continues iteratively until only simple nodes exist in the lower layer DAG graphs. Alternative multicore schedules are then created for every DAG and the program is constructed from lower to upper HTG layers in order to optimize total execution time. In other words, for every user program function, a set of alternative functions will be generated, corresponding to the alternative schedules used in the program. For the lower layers, a secondary goal is to favour larger contiguous idle time for processors.

In order to exploit the CGPEO, a user should consider the following guidelines:

- Instruction order is not important. For every layer in the HTG, the data dependences are considered and an equivalent parallel implementation of the source program is constructed. This equivalent program is optimized for execution time as well as to favour compact schedules. A set of alternative solutions is generated in order to use the most appropriate solution for the whole program.
- In order to mark a program fragment as a potential independent parallel task, create a function. Function calls are separated early, before the optimization process. The CGPEO creates a set of alternative equivalent programs for every function and from this set, the most appropriate program is selected for each function call case.
- Balance computational workload for “if” and “else” blocks. When a conditional is encountered, two alternative blocks are defined and hence, two different DAG in the HTG at a lower layer. The CGPEO will attempt to estimate how many cycles are needed for each of the blocks and for the whole conditional loop. If the workload for the two alternative paths is highly imbalanced, the execution time estimates for the node that represents them at a higher level will be inaccurate and the resulting solution will suffer from this inaccuracy. As only one of them will be used at run time, the resources required can not be estimated and a conservative approach is to allocate the maximum required resources.
- Provide realistic test inputs. Future versions for CGPEO will improve execution time estimates by feedback information from cycle-accurate simulation. The CGPEO will rely heavily on the quality of this information, and whenever a static analysis estimate fails to give result – i.e. a loop with variable number of iterations or a recursive function – the simulation feedback will be heavily utilized. In order to assist this feedback loop, as realistic test inputs as possible should be provided.

- Avoid code with difficult to measure complexity. The CGPEO tools cannot directly estimate the execution time of recursive functions and will have problems to estimate the execution time for loop structures, that do not have a fixed number of iterations. In those situations, it will fall back to use the simulator feedback heavily. If the number of iterations or the number of recursive calls varies significantly when specific input variables change, CGPEO will finally provide an optimal solution to satisfy the test inputs.
- Provide conditionals to generate alternative schedules for different input size regions. As each function call and each layer in the hierarchical task graph are scheduled individually, conditionals can guide CGPEO to consider alternative program schedules for input size regions. In addition, the input data instances should cover the alternative regions, in order for the optimizer to be able to simulate the different alternatives and obtain execution times.
- Avoid heavy computations for specific blocks. The conditional statement for an "if" block, the "start", "iteration" and "finish" statements for the "for" block and the conditional statement for the "while" block are not optimized during the CGPEO. As a result, computational intensive statements should be avoided in those statements. Another possibility is the replacement of such statements with procedure calls as the procedure call will be placed in a separate block during pre-processing and will be considered for optimization by the CGPEO.

7.1 Optimization method options and trade-offs

CGPEO provides several alternative parallelization algorithmic strategies as "*modes*". The modes are in fact combinations of heuristic and exact solution strategies. The available modes are listed in the ALMA toolchain User Guide, available in the appendix, as configuration option `gr.teimes.alma.coarsegrain.mode` in the `CoarseGrain` pass.

The key concept of the solution is the single layer in the HTG, which corresponds to a single control structure scope in the C representation of the code. For each layer, parallel schedules are produced based either on exact or heuristic methods. Exact methods, based on Mathematical Programming (specifically Integer Programming) provide provable optimal schedules, with respect to the program and cost model used, but the solution process is significantly slow. Heuristic methods apply local rules to provide scheduling decisions and are very fast, but the quality of the produced schedule is not guaranteed. Most available modes described in the ALMA tool flow manual are mixed, with a strategy to fall back to heuristic when an exact method is taking too long to respond, in order to guarantee the production of a result.

It is expected, that a user would use a mix of the above methods during development. Heuristics based methods provide performance estimates and functional correctness verification in an interactive development process. Exact methods provide programs optimized for performance and would be used for "milestone" releases (i.e. daily, or weekly).

8 Multicore Simulator

The ALMA Multicore Simulator enables the simulation of architectures specified with the ALMA System ADL. The simulator takes a System ADL file and one or multiple application files as input.

The simulator is implemented as SystemC classes representing the specified ADL modules stored in a module library. These classes can implement different simulation modes, either on behavioural level, e.g. Instruction Set Simulator (ISS), or on cycle accurate level with Cycle Accurate Simulation (CAS). The application file is then directly loaded in the simulator modules. The simulator components and inputs are presented in Figure 4.

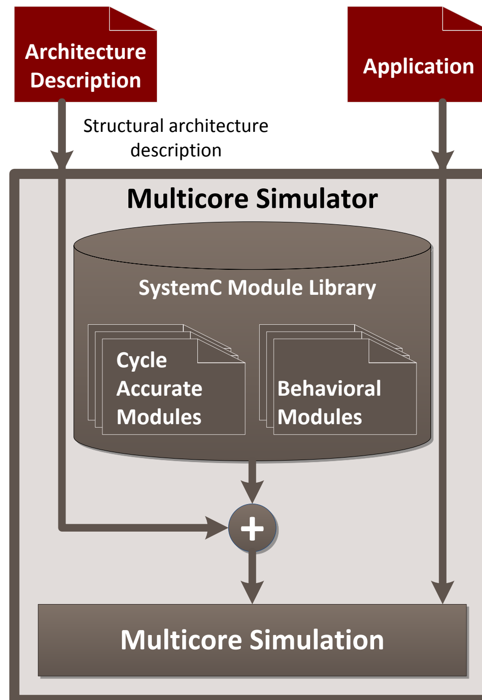

Figure 4: Multicore Simulator

8.1 Command line interface

The simulation framework can be controlled using command line parameters or configuration files. The configuration file can be used to summarize multiple command line parameters. Via special commands, the multicore framework is able to modify properties of an architecture description and forward command line arguments to specified simulation modules and instances. This feature allows the system simulation framework to forward application arguments to the core simulation modules, where the arguments can be consumed by the application.
The multicore simulator is available as command line interface utility (CLI) and is called "systemsim". The simulator is controlled by various command line arguments that are separated in a global section and a module specific section.

```
systemsim.exe --sadl <ADL-File> [#Module <elf-file> --cmd <cmd_arg>]
```

Table 8-1: Command line argument list for systemsim

8.1.1 Global Command Line Arguments

The global section describes parameters that are used in the base simulator.
The identifier "--" is used as separator in front of a command, the argument list depends on the command and is a space separated list. Therefore all commands in this section are defined as follows:

```
--cmd <cmd_arg>
```

Table 8-2: Command line argument format

Table 8-3 shows the currently available command line arguments:

Command	Argument	Description
--sadl	<ADL-file>	Specifies the path and the name of the System ADL file to use in the simulation. This argument is mandatory for every simulation run, but can be included in config-files.
--cfg	<config-files>	Specifies one or more paths to a simulator configuration file, that contains further command line arguments. Multiple files can be defined using the vector operator v().
--set	<string>	Enables the use of Data Description Language (DDL) code as preset to the specified ADL file. Useful for predefining variables.
--quiet		Turns off system simulator output.
--ssim-profile	<out-prefix>	Enables the profiling of instrumented applications. The generated Output is in JSON format.
--ssim-stats	<out-prefix>	Turns on the generation of a systemsim statistic file, containing global simulation statistics and statistics per module. The generated Output is in JSON format.
--ssim-trace	<out-prefix> [ProfilingTypeName=["Paraver Config File"]]	Generates trace files for profiling types of instrumented applications. If no configuration file is specified a default file is used. Use "ALL" as ProfilingTypeName to trace all profiling types. The generated Output is in JSON format.

Table 8-3: Available systemsim commands

8.1.2 Module Specific Command Line Arguments

Command line arguments and commands can be forwarded to modules defined in the ADL. In this way, a SystemC simulation module developer is able to define module-specific command line arguments without having to modify the simulators base modules. As identifier, "#" is used in combination with the module name within the general command line to separate modules. The format is as follows:

```
#Module_Name [arguments]
```

Table 8-4: Descriptor for module-specific command line arguments

The SystemC Module also defines the format of the module specific command argument list. All commands are forwarded to all the instances of the specified module. A parsing function in the base class of all Simulator Modules enables the correct forwarding of the parameter list to the specified module. Parameter names may begin with any character, number or special character except "#". As "#" is interpreted as separator, the definition of an escape character is necessary to enable the use of "#" as first character in arguments or options. The use of a second "#" in front of an argument is defined as such escape character. The following example shows the usage of an argument called "#argument" in a module-specific argument list.

```
#Module_Name ##argument [arguments]
```

Table 8-5: Escape character for module-specific descriptor

8.1.3 Instance-Specific Command Line Arguments

Instance-specific command line arguments work the same as module-specific with the difference that they are forwarded to the defined instance only. The Identifier used for instances is ‘@’. The format is as follows:

```
@Instance_Name [arguments]
```

Table 8-6: Descriptor for instance-specific command line arguments

An escape character is defined using two ‘@’.

```
@Instance_Name @@argument [arguments]
```

Table 8-7: Escape character for instance-specific descriptor

8.1.4 Providing Application files

The application files depend on the architecture and the implementation of the simulation class, that simulates the module the application will run on. Therefore, an application file has to be provided in the module specific section to the module that will load the application into memory.

8.2 Configuration Files

The simulator supports the use of configuration files for command line argument definition. Configuration files can be used to replace command. The command line argument “--cfg <filename>" can be used to specify a path to a configuration file that will be used instead or in combination with other configuration files or command line arguments. As the simulation framework requires at least one definition of an ADL file, either the command line or a configuration file must hold an argument for defining the ADL file.

The content of the configuration file is separated into three blocks. The "Global" block holds the global command line arguments, a "Modules" block may hold module-specific options, and an "Instances" block holds instance-specific options. The construction of a configuration file is shown in Table 8-8 below. Options do not need any prefix in configurations files. The configuration file allows the use of argument lists as vectors with the command "v(argument list)".

```
[Global] = {
   [sadl] = <ADL-FILE>;
   [cfg] = v(config-file-1, ... , config-file-n);
};
[Modules] = {
   [Module_Name] = {      //Module Specific options };
};
[Instances] = {
   [Instance_Name] = {    //Instance-specific options };
};
```

Table 8-8: Configuration file format

As multiple configuration files can be used, each block is optional. Configuration files can also be used in combination with additional command line arguments. They will be parsed and stored in a vector with the command line arguments. The options defined by configuration files will be stored at the beginning of the vector and command line arguments at the end.

8.2.1 Kahrisma Specific Options

This section provides a short overview on commonly used options for the Kahrisma architecture. The Kahrisma architecture is controlled by the Kahrisma Run Time System (RTS) Module. The command line arguments must be forwarded to the Kahrisma RTS Module using '#Kahrisma_RTS_Module'.

```
#Kahrisma_RTS_Module <elf-file> [--opt <opt_args>] [--args <arguments>]
```

The Kahrisma RTS Module supports multiple commands defined as argument list. At least one of the arguments must be a valid path to an application file in elf-file format. The position is recommended to be at the beginning of the argument list, but can be placed between options with respect to the required arguments by each option.

The available options generate information about application execution and architecture usage. Other options forward arguments to the application file or control the output of the simulator. Generally the generated information is stored in files but can be printed to the user output as described in Table 8-9.

Option	**Argument**	**Description**
--quiet		Suppress all simulator output
-		Forward all output to std:out
--args		Forward all following arguments to the application file
|	<program>	"pipe": open the defined program and redirect the output to it
--trace	<filename>	Output detailed trace-information to a file
--simpletrace	<filename>	Output a simplified trace information file optimized for human readability
--alltrace	<filename>	Output all tracing modes to multiple files
--stats	<filename>	Generate simulation statistics file
--asmstats	<filename>	Generate assembler statistics file
--funcstats	<filename>	Generate function statistics file
--allstats	<filename>	Output all available statistics to multiple files
--crashdump	<filename>	Create a long crash dump file

--callgraph	<filename>	Generate an application call-graph

Table 8-9: Kahrisma specific simulator options

The following example of the 'pipe' usage forwards the simulation output to gzip and creates a zipped output file.

```
|gzip - -o <filename>.gz
```

8.2.2 Xentium Specific Options

This section gives a brief overview of the most important options available for simulating and profiling programs on the Xentium architecture. More details can be found in the user guides for the Xentium simulator [1] and the Xentium profiler [2]. The simulator provides a number of options to generate information about the execution of a program as shown in Table 8-10. Some options will generate information directly to the user about the cycle counts, Instruction Level Parallelism (ILP) utilization and execution traces. Other options will generate profiling information that can be fed to the xentium-profiler for a more detailed account of the cycle usage and function call graph of the simulated program. Table 8-11 lists the most important options of the xentium-profiler.

Option	Argument	Description
--trace		Display trace information
--cycles		Display cycle count
--simulation-time		Display how much time the simulation took
--average-ilp		display the average Instruction Level Parallelism of the simulated program, i.e. the average number of functional units used in parallel per cycle
--trace-file	<filename>	Output trace information to file
--profile	<filename>	Generate profile information in file

Table 8-10: Extract of Xentium simulator options.

Option	Argument	Description
--callgraph		Display the call graph of the simulation
--flat-profile		Display the flat profile of the simulation

Table 8-11: Extract of Xentium profiler options.

9 ALMA flow demonstration

The ALMA tool flow components are connected using GeCoS compiler scripts. In this section, the tool components usage will be demonstrated using a simple parallel matrix multiplication code, which is written in order to expose parallelism.

The Scilab matrix multiplication code is presented in Figure 5.

```
N = 128;
N4 = int32(N/4);
A= eye(N,N);
A(1,2) = -3;
A(2,1) = %pi;

B = ones(N,N)*3 - eye(N,N);
C1 = A(N4*0+1:N4*1,:) * B;
C2 = A(N4*1+1:N4*2,:) * B;
C3 = A(N4*2+1:N4*3,:) * B;
C4 = A(N4*3+1:N4*4,:) * B;

C = [C1;C2;C3;C4];

//A
//B
//C
```

Figure 5: Parallel Matrix Multiplication Scilab Code – File `matrixmul_parallel.sce`

9.1 Front End – Generating C code

The first step in the ALMA flow is to perform source-to-source compilation from Scilab input to ALMA C code. The first line in the GeCoS script shown in Figure 6 produces C code in directory `matrixfe_output` for the file `matrixmul_parallel.sce`. The following lines include certain necessary front-end transformations to simplify the produced code by moving arrays to global scope and remove unecessary casts. The last line produces C code from the IR in the GeCoS project type variable named `project`. The simplified code is generated in the directory `simplify`.

```
project = MatrixWrapper("matrixfe_output", "matrixmul_parallel.sce");

MoveArrays2Global(project);
RemoveUnnecessaryCasts(project);
CGenerator(project,"simplify");
```

Figure 6: GeCoS Script `matrixmul_parallel.cs`: First part, Front End

9.2 Coarse and Fine Grain Parallelism Extraction

The next step of the flow is to perform loop transformations to expose coarse and fine grain parallelism. These two passes are performed as one single step, since they both use a polyhedral model and operate on SCoPs. It is not always possible to extract SCoPs after the application of coarse grain loop transformations. In such a case this would prevent fine grain transformations, therefore we apply both coarse and fine grain loop transformations at once.

The first line in Figure 7 performs this step. The first parameter is the project that is created by the earlier step. The second parameter is the number of parallel tasks that have to be extracted by the coarse grain parallelism extraction pass. The third parameter is the tile size, that has to be used by the coarse grain pass and the last parameter is the tile size that has to be used by the fine grain optimization pass.

The second line in the script generates the parallel code in the directory called CG-output.

```
UR1OptimizationFlowModule(proj,numberOfParallelTasks, cgTileSize,
fgTileSize);

Cgenerator(project,"CG-output");
```

Figure 7: GeCoS Script: Coarse grain parallelism extraction and fine grain parallelism optimization

9.3 Coarse Grain Parallelism Optimization

The coarse grain parallelism optimization consists of pre-requisite passes and two separate major passes. As the process is an iteration that involves a parallel program generation, profiling of that program and repeating with the additional knowledge of the profiling information, the coarse grain optimization process is tightly related with parallel code generation.

The main coarse grain parallelism optimization process happens in the `CoarseGrain` pass, whose responsibility is to append the profiling information of the last program execution to the annotated profiling information and to produce an optimizing parallel solution based on the profiling information, instrumented with profiling annotations for various code segments of interest. In order to initially produce profiling information, the `CoarseGrainFirstPass` produces a sequential program profiling of the sequential code.

The main parts that involve the coarse grain parallelism optimization loop are shown in Figure 8. A number of pre-requisite passes prepare the code for the coarse grain parallelism optimization. The `BBCallSplit` splits blocks that contain function calls in order to expose function calls as separate tasks, in line 4. `ProcedureDuplicator`, in line 5, duplicates procedure bodies in order to allow different schedules for each procedure call. The `ForInitSimplifier` (line 7) converts for loops that are complex for control flow duplication, annotated by `ForLoopAnalyzer`, to a variant that is like a while loop. The passes that follow in lines 8 through 14, convert the IR in SSA form to expose data dependencies and extract the Hierarchical Task Graph in the `AlmaTaskExtractor` and `ExtendTaskDependencies` passes. The coarse grain parallelism optimization passes are configured using name-value properties files, as further discussed in the ALMA toolset user manual (in the appendix). `MakespanReporter` is used to produce reports about the solution. The whole IR is saved in a file named `beforecoarse` and is loaded again after the execution of the parallel code generation, since the IR is restructured during the code generation After saving the IR before the coarse-grain optimization pass in line 16, `CoarseGrainFirstPass` produces a sequential version of the code in line 19. Lines 19-21 involve the parallel code generation and the simulator execution for profiling. The original IR, before the coarse-grain optimization passes is loaded again and `MakespanReporter` associates profiling information to IR objects. The same procedure occurs in lines 24-28, but this time with the `CoarseGrain` module that produces optimized parallel solutions. . Lines 24-28 are supposed to be executed in an iterative manner, as presented in the `coarsegrain.cs` file used in the samples. This iteration is not presented here for readability.

```
1.  p = CreateGecosProject(project_name);
2.  CDTFrontend(p);
3.
4.  BBCallSplit(p);
5.  ProcedureDuplicator(p);
6.  ForLoopAnalyzer(p);
7.  ForInitSimplifier(p);
8.
9.  ComputeSSAForm(p);
10. AlmaTaskExtractor(p,20);
11. ExtendTaskDependencies(p);
12. FixBlockNumbering(p);
13. ClearControlFlow(p);
14. BuildControlFlow(p);
15.
16. SaveGecosProject(p,"beforecoarse");
17.
18. CoarseGrainFirstPass(p, "coarsegrain-firstpass.properties");
19. ParallelCodeGeneration(p, "CodeGenInitalRun.cfg", architecture, 1);
20. RunSimulator(architecture, 1, "src-regen", "ssim_profile", 2, 2);
21.
22. p = LoadGecosProject("beforecoarse.gecosproject");
23. MakespanReporter(p,1, "coarsegrain.properties");

24. CoarseGrain(p,  numberOfProcessors, "../../coarsegrain.properties");
25. ParallelCodeGeneration(p, "CodeGenInitalRun.cfg", architecture, 1);
26. RunSimulator(architecture, 1, "src-regen", "ssim_profile", 2, 2);
27. p = LoadGecosProject("beforecoarse.gecosproject");
28. MakespanReporter(p,1, "coarsegrain.properties");
```

Figure 8: GeCoS Script demonstrating the use of CoarseGrain

The CoarseGrain pass produces Direct Acyclic Graphs (DAG) for each layer of the HTG Intermediate Representation for the C representation of the input program. Figure 9 presents the DAG for the top-level task for the `matrixmul_parallel` example, and Figure 10 presents the processor assignment for a two-processor allocation.

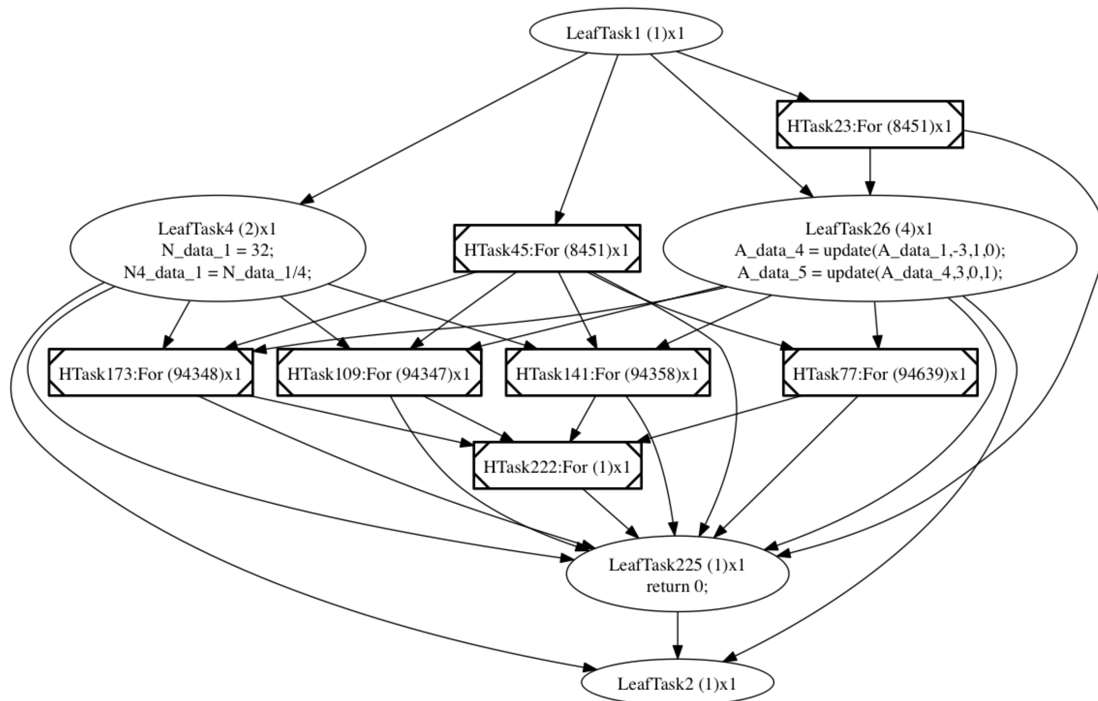

Figure 9: The top level HTG layer for `matrixmul_parallel`

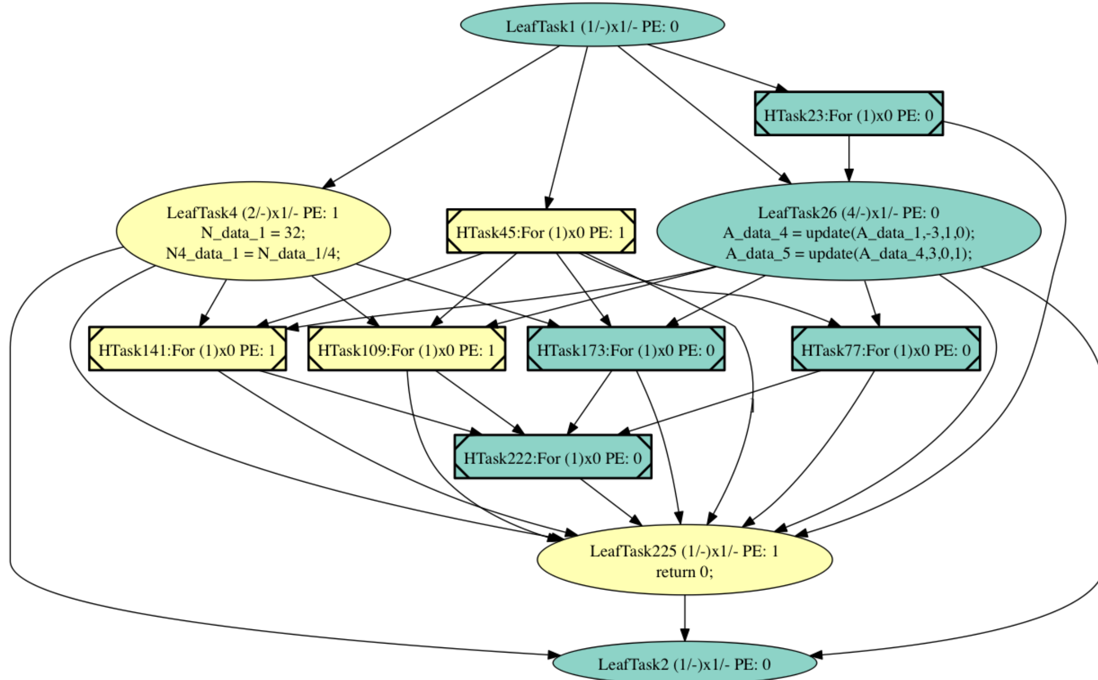

Figure 10: The parallel processor assignment for the top level HTG layer for `matrixmul_parallel` (processors are encoded as colours)

9.4 Parallel Code Generation

After the coarse-grain parallelization, the C code for the target compilers has to be created. Before the final pass `CGenerator` can be called, the intermediate representation has to be prepared for the parallel execution. After the coarse-grain passes, the application is still sequential, but each task is assigned to a specific processor. In Figure 11, line 1, the pass `CreateCommunication` resolves all data dependencies that span more than one processor. The parameter "com_mode" can be used to control the placement of communication instructions in the C code. More complex modes like "estimate" and "useSchedule" take into account the control flow of the application and the scheduling information from the coarse-grain passes to improve the execution time of the application. Simpler modes like "afterDef" should only be used if the execution time of the tool-chain is relevant; they have no other benefits.

Lines 2 and 3 handle the insertion of profiling instructions for the feedback loop to the coarse-grain pass. The options affect the search algorithms of these passes and should always be set to the same values as the passes around them. After the insertion of profiling instructions, the SSA form is removed in line 4, as it is not necessary for the next passes.

`DuplicateCFlow`, line 5, ensures that each processor has its own continuous control flow so that the application can be split into individual processes in the pass `ProcessGeneration` (line 6). The argument "architecture" here can be set to either "kahrisma" or "x2014".

Finally, the pass `CGenerator` in line 7is called which uses some extensions to generate C code for the target architectures.

```
1.  CreateCommunication(project, com_mode);
2.  AddCommProfilingInstructions(project, numberOfProcessors, 1, com_mode);
3.  AddProfilingInstructions(project, 1);
4.  RemoveSSAForm(project);
5.  DuplicateCFlow(project,numberOfProcessors);
6.  ProcessGeneration(project,numberOfProcessors,architecture);
7.  CGenerator(project,"src-regen");
```

Figure 11: GeCoS Script `matrixmul_parallel.cs`: parallel code generation

Instead of calling all the passes individually, a single `ParallelCodeGeneration` pass can be used. It can be called with different arguments, a common way can be seen in Figure 12. All options can be used as described in the individual passes.

```
ParallelCodeGeneration(project,architecture,numberOfProcessors, comMode);
CGenerator(project,"src-regen");
```

Figure 12: GeCoS Script `matrixmul_parallel.cs`: parallel code generation

9.5 Simulating the Generated Program

After the parallel code generation, the application will be simulated for performance evaluation and iterative optimization of the parallelization process. Therefore, the RunSimulator pass takes care of compiling the application source code and simulating the application. The pass expects the following parameters:

- **Architecture** *(string)* describes the architecture that will be used for execution of the application. Currently "kahrisma" and "x2014" are supported.
- **processors** *(integer)* describes the number of processors within the architecture

- **source_dir** *(string)* defines the directory with the application source files
- **command** *(string)* allows the use of architecture specific command line options, e.g. ssim_trace_kahrisma, which enables profiling and tracing of the Kahrisma architecture during simulation
- **olevel** *(integer)* defines the optimization level of the compiler
- **verbosity** *(integer)* defines the output types printed to the console. Available are “0” for no output, “1” for printing only error messages and “2” for printing error and standard out messages. Nevertheless, all output is saved to files, independent from the level of verbosity.

The call to run the simulator can be found within the coarsegrain.cs script and is defined as follows.

```
RunSimulator(architecture, processors, source_dir, command, olevel, verbosity);
```

Figure 13: GeCoS Script: compiling and simulating the application

10 Conclusions

The ALMA tool chain presents an end-to-end tool chain from a Scilab subset language directly to multicore embedded platforms. In order to remain platform agnostic, the tool chain introduces an Architecture Description Language (ADL). The tool chain consists of a front end with static code analysis optimizations, polyhedral analysis type optimizations and graph-based optimizations before the code reaches the platform specific tools. The accuracy for the optimizations is further enhanced by simulator feedback. This document describes the various tool chain components from a user point of view, provides tools demonstration and provides guidelines on how to better guide the optimization processes towards end-product performance.

11 References

1. Scilab home page: http://www.scilab.org/ [accessed 10/2/2013]
2. GeCoS Generic Compiler Suite: http://gecos.gforge.inria.fr [accessed 22/2/2013]
3. Timo Stripf, Oliver Oey, Thomas Bruckschloegl, Ralf Koenig, George Goulas, Panayiotis Alefragis, Nikolaos S. Voros, Jordy Potman, Kim Sunesen, Steven Derrien, Olivier Sentieys, and Juergen Becker. A compilation- and simulation-oriented architecture description language for multicore systems. In Computational Science and Engineering (CSE), 2012 IEEE 15th International Conference on, pages 383 – 390, Dec. 2012.
4. Timo Stripf, Oliver Oey, Juergen Becker, Gerard Rauwerda, Kim Sunesen, George Goulas, Panayiotis Alefragis, Nikolaos S. Voros, Diana Goehringer, Michael Huebner, Steven Derrien, Daniel Ménard, Olivier Sentieys, Nikolaos Kavvadias, Grigoris Dimitroulakos, Kostas Masselos, Dimitrios Kritharidis, and Nikolaos Mitas. A Flexible Approach for Compiling SciLab to Reconfigurable Multicore Embed- ded Systems. In 7th International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), York, United Kingdom, 9-11 July, 2012. (Invited Paper).

5. George Goulas, Panayiotis Alefragis, Nikolaos S. Voros,Christos Valouxis, Christos Gogos, Nikolaos Kavvadias, Grigoris Dimitroulakos, Kostas Masselos, Diana Goehringer, Steven Derrien, Daniel Ménard, Olivier Sentieys, Michael Huebner, Timo Stripf, Juergen Becker, Gerard Rauwerda, Kim Sunesen, Dimitrios Kritharidis, and Nikolaos Mitas. From scilab to multicore embedded systems: Algorithms and methodologies. In Proceedings of SAMOS XII – International Conference on Embedded Computing Systems: Architectures, Modelling and Simulation, Samos, Greece, 16-19 July, 2012. (Invited Paper).
6. Timo Stripf, Michael Huebner, Juergen Becker, Gerard Rauwerda, Kim Sunesen, George Goulas, Panayiotis Alefragis, Nikolaos S. Voros, Diana Goehringer, Steven Derrien, Daniel Ménard, Olivier Sentieys, Nikolaos Kavvadias, Kostas Masselos, and Dimitrios Kritharidis. From SciLab To High Performance Em- bedded Multicore Systems — The ALMA Approach. In Euromicro Conference on Digital System Design (DSD 2012), Cesme, Izmir, Turkey, 5-8, September 2012. (Invited Paper).
7. ALMA Consortium, Deliverable D4.2, ALMA toolset implementation (31/8/2012)
8. ALMA Consortium, Deliverable D3.2, Algorithms for intermediate code generation and hardware resources estimation (24/8/2012)
9. ALMA Consortium, Deliverable D3.5, Algorithms for Coarse-Grain Parallelism Extraction and Optimization (9/10/2013)
10. ALMA Consortium, Deliverable D3.6, Algorithms for Fine-grained Parallelism (16/9/2013)

Annex –User Manuals

The *ALMA Toolchain User Guide* and the *Matrix Frontend User Guide* documents are provided in this Annex.

ALMA Toolchain User Guide

Oliver Oey, Timo Stripf, Thomas Bruckschlögl, George Goulas

November 25, 2014

Contents

1 Toolchain Usage **2**
1.1 Project Creation . . . 2
1.2 Folder Structure . . . 2
1.3 Summary of GeCoS Compiler Scripts Options . . . 2

2 Fine Grain Parallelization **4**
2.1 UR1OptimizationFlowModule . . . 4
2.2 AlmaFloat2FixConversion . . . 5
2.3 AlmaSIMDVectorizer . . . 5

3 Coarse Grain Optimization **6**
3.1 BBCallSplit . . . 6
3.2 ForInitSimplifier . . . 7
3.3 CoarseGrainFirstPass . . . 7
3.4 CoarseGrain . . . 8
3.5 MakespanReporter . . . 8
3.6 Configuration file . . . 9
3.7 RunSimulator . . . 12

4 Parallel Code Generation **14**
4.1 CreateCommunication . . . 14
4.2 AddCommProfilingInstructions . . . 15
4.3 AddProfilingInstructions . . . 16
4.4 Duplicate Control Flow . . . 16
4.5 Process Generation . . . 16
4.6 Parallel Code Generation . . . 16

Chapter 1

Toolchain Usage

This chapter describes the usage of the ALMA toolchain starting with the Scilab/Matlab code up to the generation of the target C code for the simulator/architecture.

1.1 Project Creation

The project "‘scilab_template"’ can be used as a basis for own Scilab applications. To create a new project, copy the folder and rename all occurrences of "‘template"’ in the "‘template.cs"’ file by the project name. The source file needs to be placed in the folder "‘00_matlab_input"’. For better clarity, the *.sce and *.cs should be renamed to the project name.

To run the tool-chain, right-click on the *.cs file and choose "‘Run As → Compiler Scipt"’ in the menu.

1.2 Folder Structure

The folder "‘00_matlab_input"’ contains the used Scilab or Matlab source file. The MatrixFrontend generates the sequential C code into the folder "‘01_matrixfe_output"’. The folders "‘02_raw_output"’ and "‘03_simplify"’ contain some intermediate steps of the UR1 optimization pass which will output the C code with exposed parallelism into the folder "‘CG-output"’.

After the execution of a compiler script, the folder "‘solutions"’ contains all important files for each pass whereas the init pass is a sequential pass and all following are parallel passes.

The file "‘perfEst.html"’ that is generated directly in the project folder contains some statistic about the execution times among the different steps of the optimiuation.

1.3 Summary of GeCoS Compiler Scripts Options

When using scripts based on the template (named "‘_almaflow"’ in the WP3-integration-tests"), the following options can easily be changed:

- projectName: name of the project, should be the same name as the input file without the extension.
- sourceFile: path to the source file. Using the same structure is recommended.

- architecture: currently the two architectures "‘kahrisma"’ and "‘x2014"’ are supported.
- steps: the number of iteration steps of the coarse-grain pass with the simulator feedback loop.
- numberOfActors: the number of processors the loop tiling is targetting for. Should be the same as numberOfProcessors.
- tileSize: the target size for arrays the loop tiling is aiming for. Is used for two dimensions in the script (e.g. as 2, 2 for tileSize 2). ToDo: UR1: check if description is correct
- comMode: placement mode for the communication of the parallel code generation. "‘estimate"’ tries to minimize the the number of necessary communication by estimating the number of executions by the control structure. "‘oldImp"’ places the communication directly after the definition of the variable, but uses optimization to avoid communication inside of loops. "‘afterDef"’ is emulating the "‘oldImp"’ behavior with the "‘estimate"’ pass by assigning the lowest cost to the position after the definition.
- stacksize and ramsize: size of the RAM and the stack in Megabyte for the Kahrisma architecture. Can be kept at the default values (64 and 128) for X2014.

Chapter 2

Fine Grain Parallelization

The Fine-Grain optimizations flow provides several loop transformations exposing Sub-Word Parallelism (SWP) and enhancing data reuse. It also provides an automatic Basic-Block level SIMD Vectorizer. Furthurmore, the fine-grain optimization flow includes a floating-point to fixed-point conversion framework. All these transformations are implemented within the GeCoS framwork. It consists of the following GeCoS modules accessible through the GeCoS compiler script:

UR1OptimizationFlowModule Performs fine-grain loop transformations in addition along with coarse-grain loop transformations.

AlmaFloat2FixConversion Performs floating-point to fixed-point conversion whenever possible.

AlmaSIMDVectorizer Performs SIMD Vectorization on all basic blocks annotated with `__RLT_INSET__`.

CGeneratorSetSIMDArchitecture Set the target architecture to be used for SIMD Vectorization.

2.1 UR1OptimizationFlowModule

`UR1OptimizationFlowModule` first detects and extracts polyhedral parts (SCoP) of the application, it then applies a set of coarse and fine graine loop transformations to help exposing parallelsim and enhacing data locality and reuse. It will also automatically annotate some basic blocks with `__RLT_INSET__`. This module can be used in a GeCoS script as follows:

```
p = CreateGecosProject(project_name); # create a project
AddSourceToGecosProject(p, source_file); # add source files
CDTFrontend(p); # launch the c front-end

UR1OptimizationFlowModule(p, [unrollFactor], [CGTileSizes], [FGTileSizes])
```

All but the first argument are optional.

- [unrollFactor] should be a positive integer value.

- `CGTileSizes` should be a list of integer values representing the desired size of coarse grain tiles.
- `FGTileSizes` similar to `CGTileSizes` but for fine grain tiles.

2.2 AlmaFloat2FixConversion

To use `AlmaFloat2FixConversion`, you first need to create a float to fixed project as shown below:

```
fixProj = CreateIDFixProject(p, "output_dir"); # create a float to fix project
AlmaFloat2FixConversion(fixProj, AccuracyConstraint); # apply float to fixed conversion
```

`AccuracyConstraint` should be a string representing the maximun Noise Power allowed at the system's output

2.3 AlmaSIMDVectorizer

To perform SIMD Vectorization on a GeCoS project, the following modules should be used:

```
CGeneratorSetSIMDArchitecture(targetArchitecture); # set the target architecture
AlmaSIMDVectorizer(p); # Apply SIMD vectorization
```

Chapter 3

Coarse Grain Optimization

The Coarse-Grain Optimization involves the identification of parallelism at the Hierarchical Task Graph level. Coarse-Grain Optimization involves two main GeCoS modules, the Coarse Grain First Pass and the Coarse Grain, and several supportive modules. The GeCoS modules related to Coarse-Grain Optimization follow, in the order they appear in GeCoS compiler scripts:

BBCallSplit Separates procedure calls at the Intermediate Representation level in order for procedure calls to be available as individual atoms for parallelism.

ForInitSimplifier Converts general case of for loops (loops that are not a simple iteration over a variable without `continue` or `break` instructions in the body) into simpler structures, like `while`, that are more efficiently analyzable.

CoarseGrainFirstPass Fist Pass for coarse grain optimization, produces a sequential solution and task profiling instructions, in order to prepare for the subsequent Coarse Grain Optimization passes.

CoarseGrain The main part of Coarse Grain Optimization, produces a sequential solution and task profiling instructions, in order to prepare for the subsequent Coarse Grain Optimization passes.

MakespanReporter Collects data about the last run and produces graphs specific to the last profiled execution.

3.1 BBCallSplit

`BBCallSplit`, or *Basic Block Call Splitter*, splits basic blocks that contain procedure calls in a manner such that, a procedure is the single instruction in the block. In addition, `BBCallSplit` duplicates procedure bodies each time a new procedure call to the same procedure is met, thus allowing Coarse Grain Optimization to produce different schedules for each different case of procedure body.

`BBCallSplit` as a GeCoS module receives a `GecosProject` object as its parameter, as shown in the example below.

```
p = CreateGecosProject(project_name);
```

```
AddSourceToGecosProject(p, source_file);
CDTFrontend(p);

BBCallSplit(p);
```

3.2 ForInitSimplifier

For loops can be simple loops that iterate over a variable and not include `break` or `continue` statements. For loops that are not of the above simple, canonical form, `ForInitSimplifier` converts them into a simpler structure that exposes dependencies in the HTG.

During `ForInitSimplifier`, a loop of the general form:

```
for( {init}; {test}; {step} )
    {body}
```

Is converted to the equivalent:

```
{init}
for( ; {test};  ) {
    {body}
    {step}
}
```

`ForInitSimplifier` as a GeCoS module receives a `GecosProject` object as its parameter, as shown in the example below.

```
p = CreateGecosProject(project_name);
AddSourceToGecosProject(p, source_file);
CDTFrontend(p);

ForInitSimplifier(p);
```

3.3 CoarseGrainFirstPass

`CoarseGrainFirstPass` produces a sequential solution instrumented with task profiling annotations, in order to prepare for the subsequent Coarse Grain Optimization passes.

The `CoarseGrainFirstPass` as a GeCoS module receives a `GecosProject` object and a configuration file name as a parameter, as shown in the example below.

```
p = CreateGecosProject(project_name);
AddSourceToGecosProject(p, source_file);
CDTFrontend(p);

...

CoarseGrainFirstPass(p, "../../coarsegrain-firstpass.properties");
```

Alternatively, default values for configuration options may be used:

```
p = CreateGecosProject(project_name);
AddSourceToGecosProject(p, source_file);
CDTFrontend(p);

...

CoarseGrainFirstPass(p, "../../coarsegrain-firstpass.properties");
```

The available options for the configuration file are presented in section 3.6.

3.4 CoarseGrain

The `CoarseGrain` module is the main part of the Coarse Grain Optimization process.

It is assumed that CoarseGrain is executed after `CoarseGrainFirstPass` or `CoarseGrain` and after Parallel Code Generation and Simulator/Profiler.

The `CoarseGrain` as a GeCoS module receives a `GecosProject` object, an integer with the number of processors to use and a configuration file name as parameters.

```
CoarseGrain(p, 6, "../../coarsegrain.properties");
```

It is possible to use default parameters instead of the configuration file:

```
CoarseGrain(p, 8);
```

The default number of processors is 4:

```
CoarseGrain(p);
```

The available options for the configuration file are presented in section 3.6.

3.5 MakespanReporter

The `CoarseGrain` module provides information on the program as generated by a previous `CoarseGrain` pass and simulated in the profiler. The information includes the total running time, as well as colored DOT files according to the processor allocation.

The `MakespanReporter` as a GeCoS module receives a `GecosProject` object, an integer with the number of processors used and a configuration file name as parameters.

```
MakespanReporter(p, 6, "../../coarsegrain.properties");
```

It is possible to use default parameters instead of the configuration file:

```
MakespanReporter(p, 8);
```

The available options for the configuration file are presented in section 3.6.

3.6 Configuration file

The configuration file for CoarseGrainFirstPass, CoarseGrain and MakespanReporter has the same structure. The general format is that of Java Properties file (`java.util.Properties`). In order to provide context, the Java package or fully-qualified class name is used as property name prefix.

The available parameters are presented below:

```
gr.teimes.alma.coarsegrain.mode=4
```

The mode property dictates the solution method. In case of a random invalid mode, the `CoarseGrain` pass outputs the available modes.

The available modes are:

mode 0 Single Processor solution

mode 1 Random parallel solution

mode 2 HEFT

mode 3 MPM

mode 4 Heuristics Multisolver

mode 5 MPM MultiSover

```
seed
```

Random generator seed, in order to produce repeatable results.

```
gr.teimes.alma.coarsegrain.shouldProduceDots=true
gr.teimes.alma.coarsegrain.dotsDirectory=dot
gr.teimes.alma.coarsegrain.enableDetailedDots=true
```

Property `shouldProduceDots` controls if the module should produce DOT files in order to visualize the HTG. The dot files are produced in the `dotsDirectory` directory, which should exist before the execution of the module. The `enableDetailedDots` enables the inclusion of extra information in the DOT files.

The DOT files have special annotations in order to enable linking between Hierarchical Tasks. When the DOT files are converted to SVG and viewed in complying viewers like Mozilla Firefox, a user is able to click on Hierarchical Task references and see the corresponding DAG. The conversion from DOT to SVG can be performed using the dot processor:

```
dot -Tsvg file.dot -o file.svg
```

```
gr.teimes.alma.coarsegrain.exportDebugGraph=true
```

The `exportDebugGraph` property enables the production of a file `htg.dot` that visualizes the whole program HTG in a tree structure.

```
gr.teimes.alma.coarsegrain.SolutionExporter.disableTaskProfiling=false
gr.teimes.alma.coarsegrain.SolutionExporter.disableTaskDependencyProfiling=false
```

The above properties control the use of profiling instructions. In normal operation, those parameters are set to false.

```
gr.teimes.alma.coarsegrain.mpm.solver=ORTOOLS-SCIP
```

The solver specifies the mathematical solver used to solve subproblems. The available options are:

ORTOOLS-CBC *(default)* CBC Open Source MILP solver using Google OR-tools

ORTOOLS-GLPK GLPK Open Source Solver using Google OR-tools

ORTOOLS-SCIP SCIP Solver, Free for non-commercial and academic use, using Google OR-tools

GUROBI GUROBI commercial solvert

CPLEX IBM CPLEX commercial solver

```
gr.teimes.alma.coarsegrain.exportSVGSolution=true
gr.teimes.alma.coarsegrain.SVG.clusterHTasks = boolean
```

Property `exportSVGSolution` controls wether a visualization of the whole parallel program execution will be generated as SVG.

Property `clusterHTasks` when set enables the clustering of the tasks for a single processor in the SVG output in order to reduce visual clutter. Default value is true.

```
gr.teimes.alma.coarsegrain.mpm.timeout=6000
```

Property `timeout` limits the execution time of mathematical solver for a subproblem, time given in seconds.

```
gr.teimes.alma.coarsegrain.timeLimit=6000
```

Property `timeLimit` was intended to be a limit for the whol CoarseGrain iteration. Not used.

```
gr.teimes.alma.spmf.factory.LateAcceptanceFactory.lfa=100
gr.teimes.alma.spmf.factory.SimulatedAnnealingFactory.coolRate=0.01
gr.teimes.alma.spmf.factory.SimulatedAnnealingFactory.endTemp=0.1
gr.teimes.alma.spmf.factory.SimulatedAnnealingFactory.plateauSteps=10
gr.teimes.alma.spmf.factory.MetaheuristicFactory.timeLimit=1500
gr.teimes.alma.spmf.factory.MetaheuristicFactory.stepsLimit=1800
gr.teimes.alma.spmf.factory.MetaheuristicFactory.failedStepsLimit=11000
gr.teimes.alma.spmf.factory.SimulatedAnnealingFactory.startTemp=100.0
```

SPMF parameters. Not used.

```
Logger.gr.teimes.alma.coarsegrain.CoarseGrainCLI=INFO,
Logger.gr.teimes.alma.coarsegrain.ReturnBlockFinder=INFO,
Logger.gr.teimes.alma.coarsegrain.SymbolScales=INFO,
Logger.gr.teimes.alma.coarsegrain.CoarseGrainConfig=INFO,
Logger.gr.teimes.alma.coarsegrain.SolutionExport=INFO,
Logger.gr.teimes.alma.coarsegrain.CoarseGrainFirstPass=INFO,
Logger.gr.teimes.alma.coarsegrain.CoarseGrainPass=INFO,
Logger.gr.teimes.alma.coarsegrain.DefUseEdgeProducer=INFO,
Logger.gr.teimes.alma.coarsegrain.HTGProxy=INFO,
Logger.gr.teimes.alma.coarsegrain.MakespanReporter=INFO,
Logger.gr.teimes.alma.coarsegrain.CoarseGrain=INFO,
Logger.gr.teimes.alma.coarsegrain.TaskDependencyValidator=INFO,
```

```
Logger.gr.teimes.alma.coarsegrain.model.iterators.BottomUpHTGIterator=INFO,
Logger.gr.teimes.alma.coarsegrain.model.base.RelocatableDagSchedule=INFO,
Logger.gr.teimes.alma.coarsegrain.model.base.TaskFactory=INFO,
Logger.gr.teimes.alma.coarsegrain.model.perfestimation.PerformanceEstimator=INFO,
Logger.gr.teimes.alma.coarsegrain.model.PartialSVG=INFO,
Logger.gr.teimes.alma.coarsegrain.model.GenericSolutionSVG=INFO,
Logger.gr.teimes.alma.coarsegrain.model.ModelPackageInfo=INFO,
Logger.gr.teimes.alma.coarsegrain.model.HTGSchedule=INFO,
Logger.gr.teimes.alma.coarsegrain.model.extended.PartialHTGManipulator=INFO,
Logger.gr.teimes.alma.coarsegrain.model.ScheduleTimeline=INFO,
Logger.gr.teimes.alma.coarsegrain.model.base.MakespanEvaluator=INFO,
Logger.gr.teimes.alma.coarsegrain.model.base.DagLegalityChecker=INFO,
Logger.gr.teimes.alma.coarsegrain.model.HierarchicalTaskGraph=INFO,
Logger.gr.teimes.alma.coarsegrain.model.base.MapDagSchedule=INFO,
Logger.gr.teimes.alma.coarsegrain.model.HTGFactory=INFO,
Logger.gr.teimes.alma.coarsegrain.model.tool.SolutionIO=INFO,
Logger.gr.teimes.alma.coarsegrain.model.tool.EdgeCollectVisitor=INFO,
Logger.gr.teimes.alma.coarsegrain.model.PartialTimeline=INFO,
Logger.gr.teimes.alma.coarsegrain.model.tool.HTGConsistencyValidator=INFO,
Logger.gr.teimes.alma.coarsegrain.model.perfestimation.ProfiledRun=INFO
```

```
Logger.gr.teimes.alma.coarsegrain.heuristicsolvers.mpm.DagSolution=INFO,
Logger.gr.teimes.alma.coarsegrain.heuristicsolvers.mpm.MPLinExpr=INFO,
Logger.gr.teimes.alma.coarsegrain.heuristicsolvers.MultiSolSolverAgent=INFO,
Logger.gr.teimes.alma.coarsegrain.heuristicsolvers.mpm.SolverAgent=INFO,
Logger.gr.teimes.alma.coarsegrain.heuristicsolvers.MultiSolversStrategy=INFO,
Logger.gr.teimes.alma.coarsegrain.heuristicsolvers.mpm.SolutionConverter=INFO,
Logger.gr.teimes.alma.coarsegrain.heuristicsolvers.simple.RandomParallelSolutionGenerator=OFF,
Logger.gr.teimes.alma.coarsegrain.heuristicsolvers.mathmodeler.MathAgent=INFO,
Logger.gr.teimes.alma.coarsegrain.heuristicsolvers.mpm.MathSolver=INFO,
Logger.gr.teimes.alma.coarsegrain.heuristicsolvers.BasicDAGHeuristicSolver=INFO,
Logger.gr.teimes.alma.coarsegrain.heuristicsolvers.mpm.DagConverter=INFO,
Logger.gr.teimes.alma.coarsegrain.heuristicsolvers.mpm.RandomDag=INFO,
Logger.gr.teimes.alma.coarsegrain.heuristicsolvers.mpm.CompositeTaskScheduleAgent=INFO,
Logger.gr.teimes.alma.coarsegrain.heuristicsolvers.MultiSolTest=INFO,
Logger.gr.teimes.alma.coarsegrain.heuristicsolvers.HeuristicSolversPackageInfo=INFO
```

The above properties control the various loggers. Each logger is set to a level and a level enables all messages of this and upper levels. Each logger corresponds to a specific fully qualified java class name in the source code. The available levels are, in descending order, are:

OFF Disable all messages

ERROR Enable only error messages

WARN Enable warn messages and above

INFO Enable info messages and above

DEBUG Enable debug messages and above

FINE Enable fine messages and above

ALL Enable all messages

There are more loggers available from sub-modules as the `heuristic solvers` and `model` packages. The logger names are organized in a hierarchical manner.

```
gr.teimes.alma.coarsegrain.performanceEstimator.simulatorEnvironment=build
gr.teimes.alma.coarsegrain.performanceEstimator.simulatorJsonFilename=
simrun.profiling.stats
gr.teimes.alma.coarsegrain.performanceEstimator.datafile=perfdata.dat
```

The above options control the interface between the CoarseGrain module and the profiler. In general, they should not be modified.

The default behavior of the Coarse Grain passes is to produce the effective configuration in the `gr.teimes.alma.coarsegrain.CoarseGrainFirstPass` logger as an `INFO` level message.

3.7 RunSimulator

The module `RunSimulator` builds the executables for the target architecture and profiles them using the platform toolchain.

The `RunSimulator` as a GeCoS module receives the target architecture as a string, the number of processors as an integer, the source directory as a string, the makefile target to run as a string and verbosity as an integer, as follows.

```
RunSimulator(architecture, processors, source\_dir, command, verbosity);
```

The architecture valid options are:

"kahrisma" For Kahrisma

"x2014" For Recore X2014

The command, for kahrisma is "ssim_trace_kahrisma". For X2014, this is ignored.

The verbosity level has integer values and the result for different values makes the `RunSimulator` module to output the following things in its output:

0 quiet

1 Standard error

2 Both standard error and standard output

In any case, standard error and standard output are also saved to files for future reference.

It is also possible to define the optimization level of the compiler, using the `olevel` integer parameter. The `olevel` parameter corresponds to the compiler optimization parameter, i.e. 0 corresponds to compiler parameter `-O0`, 1 corresponds to `-O1`, etc. The optimization level is set according to the following `RunSimulator` compiler script line:

```
RunSimulator(architecture, processors, source\_dir, command,
             olevel, verbosity);
```

It is also possible to define architecture stack and RAM sizes, as follows:

```
RunSimulator(architecture, no_of_processors, sourcedir, command,
             stacksize, ramsize, olevel, verbosity);
```

For `stacksize`, `ramsize`, `olevel` and `verbosity`, a value of -1 means the default value, as specified in the Makefile for the specific architecture. Default verbosity is 1.

Chapter 4

Parallel Code Generation

The parallel code generation takes the mapped and scheduled output from the coarse-grain passes and generates C code for the target architectures. I a GeCoS script there are the following passes:

CreateCommunication Resolves the data dependencies and inserts communication functions where necessary.

AddCommProfilingInstructions Add profiling instructions to the communication functions according to the dependencies marked in the coarse-grain pass. Also generates map files to match dependency IDs to communication IDs.

AddProfilingInstructions Inserts profiling instructions for task nodes into the C code.

DuplicateCFlow Ensures that each processor has its own continuous control flow.

ProcessGeneration Creates a sequential task process for each processor.

CGenerator Uses some extensions to generate C code conforming to the ALMA API.

4.1 CreateCommunication

The pass CreateCommunication has two important functions: distribution of processor assignments and insertion of communication functions. The processor assignment takes the information from the tasks which were assigned to processors in the coarse-grain parallelization and annotates all variables, functions and blocks. All variables which are needed on more than one processor are duplicated and each processor gets its own version with the prefic "‘_pn"’ where n stands for the processor number. The communication is inserted where needed to fulfill all data dependencies. The pass is called like this:

```
CreateCommunication(p, com_mode);
```

The first parameter is a gecos project, the second is the communication mode as a string. The supported modes are:

oldImp This was the first implementation of the communication placement. The communication is placed directly after the definition of a variable. The only optimization is to check if this would place the communication inside a loop. When the usage of the variable is not in that loop, the communication will be placed after the loop. This implementation should always work but rarely finds the best placement.

estimate This is a more advanced placement. At first, all possible positions are looked for. Then, the positions are evaluated by the estimated number of times they are executed. This is done using the following scheme: a normal basic block is called once, either branch of a condition is taken half the time and a loop is called 10 times. The position with the lowest number of executions will be chosen.

afterDef This mode uses the same algorithm to search for the positions, but weighs the early positions higher so that the behavior is similar to oldImp.

useSchedule This mode uses the same algorithms as estimate to find all positions. From all positions with the lowest numbers of execution, the ones best fitting the scheduling information from the coarse-grain pass are chosen.

4.2 AddCommProfilingInstructions

This pass adds the profiling function calls to the communication according to the annotated task dependencies from the coarse-grain pass. As arguments, only the GeCoS project and the number of processors is needed. The pass also generates two files:

Profiling Map The file profiling_map.json contains all communications functions listed by their communication ID. It includes information about the sending processing element (sender_PE), the receiver PE, the corresponding annotated task dependencies, the IDs of the tasks with the definition, the usage and the transfer of the variable, the symbol name and the size of the data. An example can be seen in Listing 4.1

```
1     ['2'] = {
2     ['SenderPE'] = 0;
3     ['ReceiverPE'] = 1;
4     ['TaskDepAnno'] = v(210, 228);
5     ['DefTaskID'] = 22;
6     ['UseTaskID'] = 84;
7     ['CommTaskID'] = 18;
8     ['Symbol'] = "temp";
9     ['DataSize'] = 4;
10  };
```

Listing 4.1: Example profiling map

Dependencies The file dependencies.ddl contains information about the task dependencies. For each communication belonging to a dependency the sending PE, the task ID, the tasks including the definition and the usage of the variable as well as the sending and the receiving tasks are listed. An example can be seen in Listing 4.2.

```
1     ['219'] = {
2       ['SenderPE'] = 1;
```

```
3        ['TaskID'] = 0;
4        ['defTaskID'] = 63;
5        ['useTaskID'] = 62;
6        ['startTask'] = 88;
7        ['endTask'] = 123;
8      };
9      ['266'] = {
10       ['SenderPE'] = 1;
11       ['TaskID'] = 0;
12       ['defTaskID'] = 62;
13       ['useTaskID'] = 56;
14       ['startTask'] = 123;
15       ['endTask'] = 88;
16     };
```

Listing 4.2: Example dependencies.ddl file

4.3 AddProfilingInstructions

This pass adds the profiling instructions to all annotated tasks in the given GeCoS project. All start and end function calls are inserted into LeafTaskNodes. For annotated hierarchical task nodes, start is inserted as first instruction into the first leaf task and end as last instruction into the last leaf task.

4.4 Duplicate Control Flow

The pass DuplicateCFlow duplicates all necessary control structures so that each processor gets a continuous control flow. The only parameters necessary are the GeCoS project and the number of cores.

4.5 Process Generation

This pass distributes the control flow into tasks for the individual processors. The only necessary argument is the GeCoS project.

4.6 Parallel Code Generation

Instead of calling all passes one after the other, it is possible to only use the pass Parallel-CodeGeneration. The configuration of all individual passes can be done using a configuration file whereas the arguments architecture, number of cores and communication mode (in that order, at least with 2 arguments) can be used to override the settings in the configuration file. A sample configuration can be seen in Listing 4.3.

```
1      debug=false
2      numCores=4
3      maxPositions=5
4      defaultPE=0
5      outputDir=src-regen
6      architecture=kahrisma
7      communicationMode=estimate
```

```
8        profilingMode=all
9        singleMain=false
```

Listing 4.3: Example configuration file for parallel code generation

The settings in detail:

debug Enables a debug mode with more output.

numCores The number of processing cores.

maxPositions The maximum number of positions that should be taken into account for the communication placement. Default value is 5. Higher numbers may lead to much higher running time.

defaultPE Sets the default processing element when information was not set by coarse-grain pass or pragmas in the code.

outputDir Set the output directory where the C files should be generated.

architecture Currently only the architectures kahrisma and x2014 are supported.

communicationMode Set the mode for the placement of the communication instructions. Currently supported: oldImp, afterDef, estimate and useSchedule.

profilingMode Defines which profiling instructions should be added to the code: none, all, tasks or comm.

singleMain When enabled, all main functions for the processors are included in one single main file. Otherwise, the individual main functions are called from one main function.

Matrix Frontend User Guide

Timo Stripf

November 5, 2014

Contents

1 Input Language **6**

1.1 Language Features 6

1.1.1 [] Operator 6

1.1.2 Global Variables 6

1.1.3 Matrix Frontend Comments 6

1.1.3.1 Comments in Matlab/Scilab, Regular Code in MFE 6

1.1.3.2 Regular Code in Matlab/Scilab, Comments in MFE 7

1.2 Functions 7

1.2.1 Construct Functions 7

1.2.1.1 zeros 7

1.2.1.2 ones 8

1.2.1.3 rand 8

1.2.1.4 eye 9

1.2.1.5 diag 9

1.2.1.6 rot90 10

1.2.1.7 flip 10

1.2.1.8 reshape (Matlab) / matrix (Scilab) 11

1.2.2 Basic Functions 11

1.2.2.1 size 11

1.2.2.2 length 12

1.2.2.3 ndims 12

1.2.3 Mathematic Functions 12

1.2.3.1 sqrt 12

1.2.3.2 sum 13

1.2.3.3 prod 13

1.2.3.4 abs 14

1.2.3.5 complex 14
1.2.3.6 conj 14
1.2.3.7 real 14
1.2.3.8 imag 14
1.2.3.9 log 14
1.2.3.10 log10 15
1.2.3.11 log2 15
1.2.3.12 exp 15
1.2.3.13 ceil 15
1.2.3.14 floor 15
1.2.3.15 fix 16
1.2.3.16 round 16
1.2.3.17 mod (Matlab), modulo (Scilab) 16
1.2.3.18 sin 17
1.2.3.19 cos 17
1.2.3.20 tan 17
1.2.3.21 min 17
1.2.3.22 max 18
1.2.4 Data Type Functions 18
1.2.4.1 boolean 18
1.2.4.2 double 18
1.2.4.3 int8, int16, int32, int64 18
1.2.4.4 uint8, uint16, uint32, uint64 19
1.2.4.5 complex 19
1.2.4.6 fi 19
1.2.5 Output Functions 20
1.2.5.1 disp 20
1.2.5.2 mfe_save 21
1.2.5.3 mfe_load 21
1.2.5.4 mfe_read_image 21
1.2.5.5 mfe_write_image 22
1.2.6 Size Inference Functions 22
1.2.6.1 mfe_fixedsize 22
1.2.6.2 mfe_size 22

1.2.6.3 mfe_size_noscalar 22
1.2.6.4 mfe_dynamic 23
1.2.7 Special Functions 23
1.2.7.1 mfe_pragma 23
1.2.7.2 mfe_pragma_var 23
1.2.7.3 mfe_pragma_func 24
1.2.7.4 mfe_func_file 24
1.2.7.5 mfe_func_noinline 24
1.3 Type Inference 25
1.3.1 Static Variable States 25
1.3.2 Size Inference 26
1.3.2.1 Example: Array Access in Loops 26
1.3.2.2 Example: Redefine Variable in Loops 26
1.4 Errors, Warnings and Notes 27
1.4.1 E00001 – Error – Incompatible data types 27
1.4.2 WP00002 – Performance Warning – Consider using ceil or floor instead of fix function 27
1.4.3 WI00003 – Incompatibility Warning – Negative sqrt does not result in a complex number 27
1.4.4 WP00004 – Performance Warning – Consider using ceil or floor instead of round function 27
1.4.5 E00005 – Error – Matrix division: Cannot determine if right matrix is scalar 28
1.4.6 E00006 – Error – function $funcname: Function with $paracount parameters not supported 28
1.4.7 W00007 – Warning – Cannot check matrix multiplication consistency . . 28
1.4.8 E00008 – Error – Inconsistent matrix multiplication 29
1.4.9 E00009 – Error – Matrix multiplication not defined for hypermatrices . 29
1.4.10 E00010 – Error – Transpose is not defined for hypermatrices 29
1.4.11 E00011 – Error – One-dimensional reduction: Not supported for hypermatrices 29
1.4.12 E00012 – Error – One-dimensional reduction: Cannot detect affecting dimension 30
1.4.13 W00013 – Warning – $object has multiple references. 30
1.4.14 E00014 – Error – Variable or function $var is not defined. 30
1.4.15 E00015 – Error – Unknown constant '$name' 30

1.4.16 E00016 – Error – Invalid lvalue element 30
1.4.17 E00017 – Error – Linear array indexing not supported yet. Please rewrite a(x) to a(1,x) or a(x,1). 31
1.4.18 E00018 – Error – Multiple values on the left side of the assignment is not supported. 31
1.4.19 E00019 – Error – Array access: Cannot detect if dimension $dim is 1. . 31
1.4.20 E00020 – Error – Could not interfere shape for variable $var. 32
1.4.21 N00021 – Note – Setting variable access to fixed size for variable: '$var'. 32
1.4.22 E00022 – Error – Variable $var: Cannot assign different number of dimensions. 32
1.4.23 E00023 – Error – Invalid function type 32
1.4.24 E00024 – Error – Matrix multiplication: Cannot determine if left or right matrix is scalar 32
1.4.25 E00025 – Error – Function diag is not defined for hypermatrices 33
1.4.26 E00026 – Error – Function diag: Cannot determine if input is a vector . 33
1.4.27 E00027 – Error – $funcname function: Expecting parameter $paramno to be constant 33
1.4.28 E00028 – Error – $funcname function: Invalid parameter $paramno . . . 33
1.4.29 E00029 – Error – Lexer error: Invalid character '$char' 34
1.4.30 E00030 – Error – Variable '$var' is already global. 34
1.4.31 E00031 – Error – Global state of variable '$var' is ambiguous. 34
1.4.32 N00032 – Note – Setting variable access to interval/constant size for variable: '$var'. 34
1.4.33 E00035 – Error – Different data types for variable $var ($dt1 vs. $dt2) . 35
1.4.34 E00036 – Error – break/continue statement only possible within a loop . 35
1.4.35 W00037 – Warning – Lexer warning: Use comma instead of space to separate expressions. 35
1.4.36 E00038 – Error – MFE size of variable '$var' is ambiguous. 35
1.4.37 E00039 – Error – Syntax: Unexpected end or $ 36
1.4.38 N00040 – Note – MFE size of variable set here. 36
1.4.39 E00041 – Error – $funcname function: Expecting parameter $paramno to be a variable 36
1.4.40 E00042 – Error – $funcname function: Expecting parameter $paramno to be a string 36
1.4.41 E00043 – Error – Number of output parameters does not match to function declaration 37

1.4.42 E00044 – Error – Number of input parameters does not match to function declaration . . . 37
1.4.43 E00045 – Error – Binary operator $op: Cannot determine if left or right element could be scalar . . . 37
1.4.44 N00046 – Note – Frontend is built in debug mode ... performance could be slow! . . . 37
1.4.45 E00047 – Error – $funcname function: Expecting parameter $paramno to be a constant range . . . 38
1.4.46 E00048 – Error – Array access must match the number of dimensions of the variable. . . . 38
1.4.47 W00049 – Warning – $funcname function: Function call is deprecated . 38
1.4.48 W00050 – Warning – Constant propagation for operator '$operator' not implemented . . . 38
1.4.49 W00051 – Warning – Condition couldn't be determined . . . 38
1.4.50 W00052 – Warning – Propagation of array access result not implemented 38
1.4.51 W00053 – Warning – Variable $var not used. . . . 38
1.4.52 W00054 – Warning – Reading and writing the same variable in one statement could result in wrong behavior . . . 39
1.5 Limitations . . . 39
1.5.1 Assigning different data types to one variable . . . 39
1.5.2 Assigning different number of dimensions to one variable . . . 39
1.6 Limitations (planned to be fixed) . . . 40
1.6.1 Non-deterministic functions are duplicated . . . 40
1.6.2 Support complex numbers in matlab . . . 40
1.6.3 User defined function: Modifying a parameter inside a function affects the calling variable . . . 40
1.6.4 Reading and writing the same variable in one statement may produce wrong results . . . 40
1.6.4.1 Example 1 – Copying parts of an array . . . 41
1.6.4.2 Example 2 – Matrix multiplication . . . 41
1.6.4.3 Example 3 – Exchange . . . 41

2 Usage 42
2.1 Command Line Parameters . . . 42
2.1.1 Language . . . 42
2.2 Test Directory . . . 42

Chapter 1

Input Language

1.1 Language Features

1.1.1 [] Operator

Known limitation: Only supporting two dimensions

1.1.2 Global Variables

Global variables are supported by the frontend.

1.1.3 Matrix Frontend Comments

The Matrix Frontend support special comments that are comments in Scilab/Matlab but regular code in Matrix Frontend or vice versa. This is useful e.g. to keep compatibility of your application to Matlab/Scilab when adding matrix-frontend-specific commands. This can be used to hide any `mfe_` functions from Scilab/Matlab since these functions are only available within the Matrix Frontend.

1.1.3.1 Comments in Matlab/Scilab, Regular Code in MFE

Since the comment style within Matlab and Scilab are different, there exists also a different Matlab or Scilab syntax for the Matrix Frontend Comments for Matlab and Scilab. In Scilab code the MFE comments start with `//MFE?:` and in Matlab with `%MFE?:`. In the following example, the `mfe_fixedsize` function call is protected by an MFE comment.

Scilab

```
//MFE?: mfe_fixedsize(a);
```

Scilab

Matlab

```
%MFE?:   mfe_fixedsize(a);
```

Matlab

1.1.3.2 Regular Code in Matlab/Scilab, Comments in MFE

The MFE-comments can also be used to construct comments that are only ignored within the Matrix Frontend but not within Matlab/Scilab. The idea is to combine the MFE-comments with multi-line comments. In the following code, the debug output is only used within Scilab/Matlab.

Scilab

```
//MFE?: /*
show(a);
//MFE?: */
```

Scilab

Matlab

```
% Not supported yet
```

Matlab

1.2 Functions

1.2.1 Construct Functions

1.2.1.1 zeros

Create array of all zeros.

Syntax

`y=zeros()`	returns the scalar 0.
`y=zeros(sz1,sz2)`	creates a (m1,m2) matrix filled with zeros.
`y=zeros(sz1,az2,..,szn)`	creates a (m1,m2,..,mn) matrix filled with zeros.

Matlab

`X = zeros`	returns the scalar 0.
`X = zeros(n)`	returns an n-by-n matrix of zeros.
`X = zeros('like',p)`	returns a scalar 0 with the same data type, sparsity, and complexity (real or complex) as the numeric variable, p.
`X = zeros(n,'like',p)`	returns an n-by-n array of zeros like p.
`X = zeros(sz1,...,szN,'like',p)`	returns an sz1-by..-by-szN array of zeros like p.
`x = zeros(sz,'like',p)`	returns an array of zeros like p where the size vector, sz, defines size(X).

Matlab

Scilab

`X = zeros(n)`	for a matrix of same size of A.

Scilab

Arguments

`sz1,...,szN` Two or more integer specifying the size of the dimension.

1.2.1.2 ones

Create array of all Ones.

It creates arrays of all ones on behalf of zeros in the section Please refer to section 1.2.1.1 for documentation of the syntax.

1.2.1.3 rand

Uniformly distributed pseudorandom numbers

It creates array of Uniformly distributed pseudorandom numbers on behahalf of zeros and operational syntaxes are same as zeros.

Please refer to section 1.2.1.1 for documentation of the syntax.

1.2.1.4 eye

eye Identity matrix.

`X = eye()`	eye() produces a identity matrix with undefined dimensions. Dimensions will be defined when this identity matrix is added to a matrix with fixed dimensions.
`eye(n,m)`	returns an n-by-m matrix with ones on the main diagonal and zeros elsewhere.

Matlab

`X = eye`	eye returns the scalar, 1.
`X = eye(n)`	returns an n-by-n identity matrix with ones on the main diagonal and zeros elsewhere.
`X = eye(sz)`	returns an array with ones on the main diagonal and zeros elsewhere. The size vector, sz, defines size(I). For example, eye([2,3]) returns a 2-by-3 array with ones on the main diagonal and zeros elsewhere.
`X = eye('like',p)`	returns a scalar, 1, with the same data type, sparsity, and complexity (real or complex) as the numeric variable, p.
`X = eye(n,'like',p)`	returns an n-by-n identity matrix like p.
`x = eye(sz,'like',p)`	returns a matrix like p where the size vector, sz, defines size(I).

Matlab

Scilab table syntax

`X = eye(n)`	According to its arguments defines an mxn matrix with 1 along the main diagonal or an identity matrix of the same dimension as A . Caution: eye(10) is interpreted as eye(A) with A=10 i.e. 1. (It is a ten by ten identity matrix!)If A is a linear system represented by a syslin list, eye(A) returns an eye matrix of appropriate dimension: (number of outputs x number of inputs).

Scilab table syntax

1.2.1.5 diag

Get diagonal elements or create diagonal matrix

`D = diag(v,k)`	places the elements of vector v on the kth diagonal. k=0 represents the main diagonal, k>0 is above the main diagonal, and k<0 is below the main diagonal.

Matlab table syntax

`D = diag(v)`	returns a square diagonal matrix with the elements of vector v on the main (k=0) diagonal.
`x = diag(A)` `x = diag(A,k)`	returns a column vector of the main (k=0) diagonal elements of A. diag(A,k) returns a column vector of the elements on the kth diagonal of A.

Matlab table syntax

1.2.1.6 rot90

Rotate array 90 degrees

Matlab table syntax

`B = rot90(A)`	rot90(A) rotates array A counterclockwise by 90 degrees. For multi-dimensional arrays, rot90 rotates in the plane formed by the first and second dimensions.
`B = rot90(A,k)`	rotates array A counterclockwise by k*90 degrees, where k is an integer.

Matlab table syntax

1.2.1.7 flip

Flip order of elements

Matlab table syntax

`B = flip(A)`	returns array B the same size as A, but with the order of the elements reversed. The dimension that is reordered in B depends on the shape of A: If A is vector, then flip(A) reverses the order of the elements along the length of the vector. If A is a matrix, then flip(A) reverses the elements in each column. If A is an N-D array, then flip(A) operates on the first dimension of A in which the size value is not 1.
`B = flip(A,dim)`	reverses the order of the elements in A along dimension dim. For example, if A is a matrix, then flip(A,1) reverses the elements in each column, and flip(A,2) reverses the elements in each row.

Matlab table syntax

1.2.1.8 reshape (Matlab) / matrix (Scilab)

Reshape array

Matlab table syntax

`B = reshape(A,m,n)`	returns the m-by-n matrix B whose elements are taken column-wise from A. An error results if A does not have m*n elements.
`B = reshape(A,m,n,p,...)`	returns an n-dimensional array with the same elements as A but reshaped to have the size m-by-n-by-p-by-.... The product of the specified dimensions, m*n*p*..., must be the same as numel(A).
`B = reshape(A,[m n p ...])`	returns an n-dimensional array with the same elements as A but reshaped to have the size m-by-n-by-p-by-.... The product of the specified dimensions, m*n*p*..., must be the same as numel(A).
`B = reshape(A,...,[],...)`	calculates the length of the dimension represented by the placeholder [], such that the product of the dimensions equals numel(A). The value of numel(A) must be evenly divisible by the product of the specified dimensions. You can use only one occurrence of [].

Matlab table syntax

Scilab syntax

`y=matrix(v,n,m)`	For a vector or a matrix with n x m entries, the command y=matrix(v,n,m) or similarly y=matrix(v,[n,m]) transforms the v vector (or matrix) into an nxm matrix by stacking columnwise the entries of v.
`y=matrix(v, sizes)`	For an hypermatrix such as prod(size(v))==prod(sizes), the command y=matrix(v,sizes) (or equivalently y=matrix(v,n1,n2,...nm)) transforms v into an matrix or hypermatrix by stacking "columnwise" (first dimension is varying first) the entries of v. y=matrix(v,sizes) results in a regular matrix if sizes is a scalar or a 2-vector.

Scilab syntax

1.2.2 Basic Functions

1.2.2.1 size

an array or n-D array (constant, polynomial, string, boolean, rational)

`b=size(a)`	it returns b = [3 2] as the size of the matrix a.

Matlab table syntax

`[m,n] = size(X)`	returns the size of matrix X in separate variables m and n.
`m = size(X,dim)`	returns the size of the dimension of X specified by scalar dim.

Matlab table syntax

Scilab table syntax

`y=size(x [,sel])`
 	
`[nr,nc]=size(x)`
	a matrix (constant, polynomial, string, boolean, rational) x, with only one lhs argument size returns a 1x2 vector [number of rows, number of columns]. Called with LHS=2,
	returns nr,nc = [number of rows, number of columns]. sel may be used to specify what dimension to get.

Scilab table syntax

1.2.2.2 length

Length of vector or largest array dimension

`N=length(m)`	returns the number of elements along the largest dimension of an array. array is an array of any MATLAB data type and any valid dimensions. number Of Elements is a whole number of double class.

1.2.2.3 ndims

Number of array dimensions

`N = ndims(A)`	returns the number of dimensions in the array A. The number of dimensions is always greater than or equal to 2. The function ignores trailing singleton dimensions, for which size(A,dim) = 1.

1.2.3 Mathematical Functions

1.2.3.1 sqrt

Square root

`y=sqrt(r)`	returns the square root of each element of the array X.
`y=sqrt(c)`	returns the square root of each element of the array X. For the elements of X that are negative or complex, sqrt(X) produces complex results.

1.2.3.2 sum

Sum of array elements

`y=sum(x)`	returns in the scalar y the sum of all the elements of x.
`S = sum(A,dim)`	sum(A,dim) sums the elements of A along dimension dim. The dim input is a positive integer scalar.

Matlab syntax

`S = sum(___,type)`	accumulates in and returns an array in the class specified by type, using any of the input arguments in the previous syntaxes. type can be 'double' or 'native'.

Matlab syntax

Scilab syntax

`y=sum(x,outtype)`	The outtype argument rules the way the summation is done.

Scilab syntax

1.2.3.3 prod

Product of array elements

`y=prod(x)` `S = prod(A,dim)`	prod(A) returns the product of the array elements of A. prod(A,dim) returns the products along dimension dim. For example, if A is a matrix, prod(A,2) is a column vector containing the products of each row.

Matlab syntax

`S =prod(___,type)`	and returns an array in the class specified by type, using any of the input arguments in the previous syntaxes. type can be 'double' or 'native'.

Matlab syntax

Scilab syntax

`y=prod(x,outtype)`	The outtype argument rules the way the summation is done.

Scilab syntax

1.2.3.4 abs

absolute value, magnitude

`B = abs(r)`	returns the absolute value of the elements of x.
`B = abs(c)`	returns the absolute value of the elements of x. When x is complex, abs(x) is the complex modulus (magnitude) of the elements of x.

1.2.3.5 complex

Create complex array

`z = complex(x)`	returns the complex equivalent of x, such that isreal(z) returns logical 0 (false).
`z = complex(a,b)`	creates a complex output, z, from two real inputs, such that z = a + bi.

1.2.3.6 conj

Complex conjugate

`C = conj(Z)`	returns the complex conjugate of the elements of Z.

1.2.3.7 real

Real part of complex number

`C = real(Z)`	returns the real part of the elements of the complex array Z.

1.2.3.8 imag

Imaginary part of complex number

`C = imag(Z)`	returns the imaginary part of the elements of array Z.

1.2.3.9 log

Natural logarithm

`Y = log(r)`	returns the natural logarithm of each element in array X. The function accepts real inputs. For real values of X in the interval (0, Inf), log returns real values in the interval (-Inf ,Inf).
`Y = log(c)`	returns the natural logarithm of each element in array X. The function accepts both real and complex inputs. For real values of X in the interval (0, Inf), log returns real values in the interval (-Inf ,Inf).

1.2.3.10 log10

logarithm

`Y = log10(r)`	returns the base 10 logarithm of the elements of X.
`Y = log10(c)`	returns the base 10 logarithm of the elements of X. The function accepts both real and complex inputs. For real values of X in the interval (0, Inf), log10 returns real values in the interval (-Inf ,Inf).

1.2.3.11 log2

Base 2 logarithm and dissect floating-point numbers into exponent and mantissa

`y=log2(x)`	computes the base 2 logarithm of the elements of X.
`Y = log2(c)`	returns the base 2 logarithm of the elements of X. The function accepts both real and complex inputs. For real values of X in the interval (0, Inf), log10 returns real values in the interval (-Inf ,Inf).

1.2.3.12 exp

element-wise exponential

`Y = exp(r)`	returns the exponential for each element of array X. The function accepts real and inputs. For real values of X in the interval (-Inf, Inf), exp returns real values in the interval (0,Inf).
`Y = exp(c)`	returns the exponential for each element of array X. The function accepts both real and complex inputs. For real values of X in the interval (-Inf, Inf), exp returns real values in the interval (0,Inf).

1.2.3.13 ceil

Round toward positive infinity

`B = ceil(A)`	rounds the elements of A to the nearest integers greater than or equal to A. For complex A, the imaginary and real parts are rounded independently. Special case: where front end does not support in complex numbers

1.2.3.14 floor

round down

`B = floor(r)`	rounds the elements of A to the nearest integers less than or equal to A.
`B = floor(c)`	rounds the elements of A to the nearest integers less than or equal to A. For complex A, the imaginary and real parts are rounded independently.

1.2.3.15 fix

Round toward zero

`B = fix(r)`	rounds the elements of A to the nearest integers less than or equal to A.
`B = fix(c)`	rounds the elements of A to the nearest integers less than or equal to A. For complex A, the imaginary and real parts are rounded independently.

1.2.3.16 round

rounding

y=round(x)

round(x) rounds the elements of x to the nearest integers.

`y=round(r)`	rounds the elements of X to the nearest integers. Positive elements with a fractional part of 0.5 round up to the nearest positive integer. Negative elements with a fractional part of -0.5 round down to the nearest negative integer.
`y=round(c)`	rounds the elements of X to the nearest integers. Positive elements with a fractional part of 0.5 round up to the nearest positive integer. Negative elements with a fractional part of -0.5 round down to the nearest negative integer. For complex X, the imaginary and real parts are rounded independently.

1.2.3.17 mod (Matlab), modulo (Scilab)

Modulus after division

Matlab syntax

`B = mod(x,y)`	returns the modulus after division of X by Y.
`B = rem(x,y)`	returns the remainder after division of X by Y.

Matlab syntax

Scilab syntax

`B = modulo(x,y)`	returns the modulus after division of X by Y.
`B = pmodulo(x,y)`	returns the positive arithmetic remainder modulo B.

Scilab syntax

1.2.3.18 sin

Sine of argument in radians

Syntax

`Y = sin(r)`	returns the sine of the elements of X. The sin function operates element-wise on arrays.
`Y = sin(c)`	returns the sine of the elements of X. The sin function operates element-wise on arrays. The function accepts both real and complex inputs. For real values of X in the interval [-Inf, Inf], sin returns real values in the interval [-1 ,1]. For complex values of X, sin returns complex values. All angles are in radians

1.2.3.19 cos

Cosine of argument in radians

Syntax

Please refer to section 1.2.3.18 for documentation of the syntax.

1.2.3.20 tan

[t]=tan(x)

The elements of t are the tangent of the elements of x

Syntax

Please refer to section 1.2.3.18 for documentation of the syntax.

1.2.3.21 min

Smallest elements in array

`m=min(A , B)`	returns A for two parameter version for a real vector or matrix.

Matlab syntax

`C = min(A,[],dim)`	returns the smallest elements along the dimension of A specified by scalar dim. For example, min(A,[],1) produces the minimum values along the first dimension of A.
`C = [C,I] = min(...)`	finds the indices of the minimum values of A, and returns them in output vector I. If there are several identical minimum values, the index of the first one found is returned.

Matlab syntax

Scilab syntax

`[m [,k]]=min(A,'c')`	For A, a real vector or matrix, min(A) is the largest element A. [m,k]=min(A) gives in addition the index of the minimum. A second argument of type string 'r' or 'c' can be used : 'r' is used to get a row vector m such that m(j) contains the minimum of the j th column of A (A(:,j)), k(j) gives the row indice which contain the minimum for column j. 'c' is used for the dual operation on the rows of A. 'm' is used for compatibility with Matlab.
`[m [,k]]=min(A1,A2,...,An)`	where all the Aj are matrices of the same sizes,returns a vector or a matrix m of size size(m)=size(A1) such that m(i)= min(Aj(i)), j=1,...,n.

Scilab syntax

1.2.3.22 max

Largest elements in array

Same as the min syntax but it returns largest elements in an array behalf of smallest array in an array.

1.2.4 Data Type Functions

1.2.4.1 boolean

Boolean evaluation

MATLAB Compatibility:

This functionality does not run in Matlab

Scilab Compatibility:

Scilab Objects, boolean variables and operators

1.2.4.2 double

1.2.4.3 int8, int16, int32, int64

Convert to n-bit signed integer, where n=8,16,32,64

`z = int8(x)`	converts the elements of an array into signed 8-bit (1-byte) integers of class int8.
`z = int16(x)`	converts the elements of an array into signed 16-bit (2-byte) integers of class int16.
`z = int32(x)`	converts the elements of an array into signed 32-bit (4-byte) integers of class int32.
`z = int64(x)`	converts the elements of an array into signed 64-bit (8-byte) integers of class int64.

1.2.4.4 uint8, uint16, uint32, uint64

Convert to n-bit unsigned integer, where n=8,16,32,64.

`z = uint8(x)`	converts the elements of an array into unsigned 8-bit (1-byte) integers of class uint8.
`z = uint16(x)`	converts the elements of an array into unsigned 16-bit (2-byte) integers of class uint16.
`z = uint32(x)`	converts the elements of an array into unsigned 32-bit (4-byte) integers of class uint32.
`z = uint64(x)`	converts the elements of an array into unsigned 64-bit (8-byte) integers of class uint64.

1.2.4.5 complex

create a complex number

`z = complex(x)`	returns the complex equivalent of x, such that is real(z) returns logical 0 (false).
`z = complex(a,b)`	creates a complex output, z, from two real inputs, such that z = a + bi.

1.2.4.6 fi

create a fixed-point number

`a = fi`	is the default constructor and returns a fi object with no value, 16-bit word length, and 15-bit fraction length.
`a = fi(v)`	returns a signed fixed-point object with value v, 16-bit word length, and best-precision fraction length.
`a = fi(v,s)`	returns a fixed-point object with value v, Signed property value s, 16-bit word length, and best-precision fraction length. s can be 0 (false) for unsigned or 1 (true) for signed.
`a = fi(v,s,w)`	returns a fixed-point object with value v, Signed property value s, word length w, and best-precision fraction length.
`a = fi(v,s,w,f)`	returns a fixed-point object with value v, Signed property value s, word length w, and fraction length f. Fraction length can be greater than word length or negative.
`a = fi(...,F)`	returns a fixed-point object with a fimath object F.
`a = fi(...,T)`	returns a fixed-point object with a numerictype object T.
`a = fi(...,'PropertyName',PropertyValue...)`	allow you to set fixed-point objects for a fi object by property name/property value pairs.

1.2.5 Output Functions

1.2.5.1 disp

Displays variable

Matlab syntax

`disp(X)`	Disp(X) displays the contents of X without printing the variable name. Disp does not display empty variables

Matlab syntax

Scilab equivalent table syntax

`disp(x1,[x2,...xn])`	displays xi with the current format. xi s are arbitrary objects (matrices of constants, strings, functions, lists, ...)

Scilab equivalent table syntax

1.2.5.2 mfe_save

This function saves a variable to a file. The variable can be loaded again using the `mfe_load` function.

`mfe_save('filename', var)`	Save variable to file)

1.2.5.3 mfe_load

This function loads a variable from a file created by `mfe_save`. The variable to load to has to be initialized with the correct dimension sizes. The variable has to be of type `double`.

`mfe_load('filename', var)`	Load variable from file

Example:

```
a = double(zeros(10, 10));
mfe_load('file.dat', a);
```

1.2.5.4 mfe_read_image

This function reads an image from a file. Supported formats by this function are jpg, png and bmp. The width and height must be provided as additional parameter. The image can be optionally loaded as greyscale.

The type of the resulting variable is uint8. In greyscale mode, the resulting variable has 2 dimensions. In colored mode, the resulting variable has 3 dimensions and the size of 3rd dimension is 3 to contain the color channels.

The size of the loaded image must be provided in the 2nd and 3rd parameter. Each of the parameter could be either a constant or a constant range. If the loaded image has a different shape than provided by the parameters, the behavior of the function is undefined.

`mfe_read_image('filename', xsize, ysize)` `mfe_read_image('filename', xsize, ysize, 'Greyscale')`	Load colored image from file Load greyscale image from file

Example:

```
a = mfe_read_image('test.bmp', 10:64, 10:64);
```

1.2.5.5 mfe_write_image

This function writes an image into a file. Supported formats by this function are jpg, png and bmp.

The data type of the variable is irrelevant. If the variable has 2 dimensions, a greyscale image is saved. If the variable has 3 dimensions, the 3rd dimension contains the color parts and a colored image is saved.

`mfe_write_image('filename', var)`	Write image to file

Example:

```
var = rand(100, 50) * 255;
mfe_write_image("test.bmp", var);
```

1.2.6 Size Inference Functions

1.2.6.1 mfe_fixedsize

This function indicates that the size of variable is not changed anymore. This is helpful to hint the frontend that any array accesses to the variable won't affect their size.

`mfe_fixedsize(var)`	Set variable into fixed size mode

1.2.6.2 mfe_size

This function sets the size of the dimensions of a variable to intervals and/or constant values.

`mfe_size(var, dim_1, ..., dim_n)`	Set size of one variable

Example:

```
mfe_size(a, 3, 3, 1:10);
```

1.2.6.3 mfe_size_noscalar

This functions sets the size of the dimensions of a variable to intervals and/or constant values. Additionally, it indicates that the variable should never interpreted as a scalar. That is especially for the type inference system important. The operator type often changes if one expression is a scalar or not. For growing variables, the size of a variable could be theoretically a scalar (1x1) and that could cause the operator type to become undecidable without this function.

`mfe_size_noscalar(var, dim_1, ..., dim_n)`	Set size of one variable and declare to never thread it as a scalar

Example:

```
mfe_size_noscalar(a, 1:10, 1:10);
```

1.2.6.4 mfe_dynamic

This function indicates that the size of a variable should be automatically calculated by the frontend. That is the default mode of a variable. It can be used to reset any previous setting from other size inference functions.

`mfe_dynamic(var)`	Set variable to dynamic mode.

1.2.7 Special Functions

1.2.7.1 mfe_pragma

This functions added pragmas to Matlab or Scilab statements. In the C output code, a `#pragma` directive is added before the statement code.

`mfe_pragma(string)`	Adds a parama to the successive directive.

Example

The following Matlab code

```
mfe_pragma('omp parallel for');
a = zeros(3,3);
```

adds the pragma to the C code:

```
#pragma omp parallel for
for (v1 = 0; v1 < 3; ++v1) {
  for (v0 = 0; v0 < 3; ++v0) {
    a_data[v1][v0] = 0;
  }
}
```

1.2.7.2 mfe_pragma_var

This functions added pragmas to Matlab or Scilab variables. In the C output code, a `#pragma` directive is added before the variable declaration.

`mfe_pragma_var(var,str)`	Adds a pragma to the variable.

1.2.7.3 mfe_pragma_func

This functions added pragmas to Matlab or Scilab variables. In the C output code, a `#pragma` directive is added before the function declaration.

`mfe_pragma_func(var)`	Adds a pragma to the containing function.

1.2.7.4 mfe_func_file

This special directive controls the C output file name for the function that contains the directive. Per default, each function is written to the output C file specified by the command line parameter.

`mfe_func_file(filename)`	Set the output file name for the containing function.

Example

```
function [out1] = f()
   mfe_func_file('func_f.c');

   out1 = rand();
end

out1 = f();
```

1.2.7.5 mfe_func_noinline

This special directive controls deactivates inlining for the function that contains the directive. No constant propagation is performed over the function boundary if noinline is specified.

`mfe_func_noinline()`	Deactivates inlining for the containing function.

Example

```
function [out1] = f()
   mfe_func_noinline();

   out1 = 2;
end

a = f()

for i = 1:a
   i
end
```

Prevents the variable a to be replaced by 2.

1.3 Type Inference

The Matlab or Scilab language is a scripting language that features dynamic typing. That means that the types and operator are checked at run time and are not fixed. For an efficient compilation, it is important that the data types and operators are static and known at compile time.

Therefore, the Matrix Frontend comprises an advanced type inference engine that tries to calculate the data type and maximum size for each variable. If the type inference fails, an error message is generated. In that case, the user can annotate additional type information to fix the problem.

1.3.1 Static Variable States

The variable states comprises several flags of a variable:

- Global state (set by Global statement)
- Size state (set by size inference functions, see 1.2.6)

The flags are propagated within a function through the possible control flow. The control flow refers to the order in which the individual statements are executed. For every variable usage, the state of a variable must be non-ambiguous.

In the following example, the variable is changed to global within an if-then-else statement. After the statement, the global state of the variable is ambiguous. It depends on the run-time value of x whether the variable a is global or not. So it is at compile-time undecidable and would produce an error.

```
a = 1;    % Local (default at beginning of a function)
if (x)
   a = 2;    % Local
   global a;
   a = 3;    % Global
else
   a = 4;    % Local
end
a = 5;    % Ambiguous
```

In the following example, the variable is changed to global within the for loop. At the beginning of the for loop, the variable is local in the first iteration and global in any other iteration, and therefore ambiguous. After the for loop, it is also ambiguous since a for loop could be executed 0 times.

```
a = 1;    % Local (default at beginning of a function)
for i = 1:10
   a = 2;    % Ambiguous
   global a;
   a = 3;    % Global
end
a = 4;    % Ambiguous
```

1.3.2 Size Inference

During size inference, the shape of a matrix is calculated.

1.3.2.1 Example: Array Access in Loops

In this example, the maximum size of variable `a` cannot be interfered. The maximum size of `a` depends on the maximum value of `i2` and that could currently not calculated.

```
i2=1;
a=[];
for i=1:10
   a(1,i2) = i;
   i2 = i2 + 1;
end
```

One solution to fix the problem is the `mfe_fixedsize` function. This function indicates that the size of variable `a` is not changed after initialization.

```
i2=1;
a=zeros(1,10);
%MFE?: mfe_fixedsize(a);
for i=1:10
   a(1,i2) = i;
   i2 = i2 + 1;
end
```

1.3.2.2 Example: Redefine Variable in Loops

TODO: Problem of this example

```
a = zeros(16, 16);

for i=1:1:3
      b = size(a, 1)
      a = zeros(b/2, 16);
end
```

One Solution to fix this problem is the `mfe_size` function. This function sets the size of variable `a` to a defined interval or constant.

```
a = zeros(16, 16);

for i=1:1:3
      %MFE?: mfe_size(a, 2:16, 16);
      b = size(a, 1)
      a = zeros(b/2, 16);
end
```

1.4 Errors, Warnings and Notes

1.4.1 E00001 – Error – Incompatible data types

The data types are not compatible and cannot be combined.

Example

```
10 + 'xx';
```

1.4.2 WP00002 – Performance Warning – Consider using ceil or floor instead of fix function

The performance of the fix function is worse in contrast to ceil or floor. This warning is raised when the fix function is used.

Example

```
fix(4.4);
```

1.4.3 WI00003 – Incompatibility Warning – Negative sqrt does not result in a complex number

The warning is automatically raised, when a sqrt is used.

Example

```
sqrt(1);
```

To make your code Matlab and Scilab compatible, please use

```
real(sqrt(1))
```

for a real square root and

```
sqrt(complex(1))
```

to perform a complex square root.

1.4.4 WP00004 – Performance Warning – Consider using ceil or floor instead of round function

The performance of the round function is worse in contrast to ceil or floor. This warning is raised when the round function is used.

Example

```
round(4.4);
```

1.4.5 E00005 – Error – Matrix division: Cannot determine if right matrix is scalar

It must be clear at compile time, if the right side of a matrix division is scalar or not. The error indicates that this is not decidable by the front end.

Example

```
a = rand(3,3)
b = rand(1,1)
mfe_size(b, 1:2, 1:2)
a / b
```

1.4.6 E00006 – Error – function $funcname: Function with $paracount parameters not supported

The error indicates, that the built-in function with a specific number of parameters is not supported.

Example

```
sqrt(10,10,10)
```

1.4.7 W00007 – Warning – Cannot check matrix multiplication consistency

The matrix multiplication is defined only, if the number of columns in the left matrix is equal to the number of rows in the right matrix. If that is not decidable at compile time, a warning is raised to indicate a potential run-time error.

Example

```
a = rand(3,3);
b = rand(3,3);
mfe_size(b, 1:3, 3);
a*b;
```

1.4.8 E00008 – Error – Inconsistent matrix multiplication

The matrix multiplication is defined only, if the number of columns in the left matrix is equal to the number of rows in the right matrix. The error is raised if the condition is not met.

Example

```
rand(3,3) * rand(2,2)
```

1.4.9 E00009 – Error – Matrix multiplication not defined for hypermatrices

The matrix multiplication is defined only for 2 dimensional matrices and not for hypermatrices. The error is raised if it is used on hypermatrices.

Example

```
rand(3,3,3) * rand(3,3,3)
```

1.4.10 E00010 – Error – Transpose is not defined for hypermatrices

The transpose operator is defined only for 2 dimensional matrices and not for hypermatrices. The error is raised if it is used on hypermatrices.

Example

```
rand(3,3,3)'
```

1.4.11 E00011 – Error – One-dimensional reduction: Not supported for hypermatrices

The reduction into one dimension (max, min, sum, prod) is not supported for hypermatrices right now.

Example

```
b = max(rand(3,3,3))
```

1.4.12 E00012 – Error – One-dimensional reduction: Cannot detect affecting dimension

The reduction into one dimension (max, min, sum, prod) is performed into the first dimension that is not 1. The detection of the dimension must be compile time constant. The error is raised if it cannot be determined.

Example

```
a = rand(3,3);
mfe_size(a, 1:3, 3);
max(a)
```

1.4.13 W00013 – Warning – $object has multiple references.

Internal error, please report to development team.

1.4.14 E00014 – Error – Variable or function $var is not defined.

A variable is used but not defined. For each variable usage, there must be a definition on any possible control-flow path.

Example

```
if (rand())
   a = 1;
end
b = a
```

In this example, the variable a is defined only if the condition is met.

1.4.15 E00015 – Error – Unknown constant '$name'

Scilab only: A unknown constant is used.

Example

```
%xyz
```

1.4.16 E00016 – Error – Invalid lvalue element

A lvalue is the term on the left side of a equal sign. On the left side, only variables and array of variables are allowed. Any other expression will result in this error.

Example

```
rand(3,3) = 10;
```

1.4.17 E00017 – Error – Linear array indexing not supported yet. Please rewrite a(x) to a(1,x) or a(x,1).

Linear index of matrices is not supported yet. This will be added in future.

Example

```
a = rand(3,3);
a(1)
```

1.4.18 E00018 – Error – Multiple values on the left side of the assignment is not supported.

Multiple values on the left side of the assignment for buildin functions is not supported yet. This will be added in future.

Example

```
a = rand(3,3);
[x,y] = size(a);
```

This example show one typical case that uses multiple values on the left side. This can be easily rewritten to

```
a = rand(3,3);
x = size(a,1);
y = size(a,2);
```

1.4.19 E00019 – Error – Array access: Cannot detect if dimension $dim is 1.

For matrix indexing, it is sometimes required to decide if the access to one dimension has one element. If that is not decidable at compile time, an error is raised.

Example

```
a = 1:10;
mfe_size(a,1:10,1);
b = rand(10,10,10);
b(1,1,a)
```

1.4.20 E00020 – Error – Could not interfere shape for variable $var.

Example

```
a = [];
x = 1;
for i=1:10
   a(x,1) = i;
   x = x+1;
end
```

1.4.21 N00021 – Note – Setting variable access to fixed size for variable: '$var'.

A note is output for every variable access that is set to fixed size.

Example

```
mfe_fixedsize(a);
a = rand(3,3);
a(2,2) = 1;
```

1.4.22 E00022 – Error – Variable $var: Cannot assign different number of dimensions.

It is not possible to assign arrays with different numbers of dimensions to one variable. See limitation in section 1.5.2;

Example

```
a = zeros(3,3);
a = zeros(1,2,3);
```

1.4.23 E00023 – Error – Invalid function type

Syntax error.

1.4.24 E00024 – Error – Matrix multiplication: Cannot determine if left or right matrix is scalar

It must be clear at compile time, if the right side or left side of a matrix multiplication is scalar. The error indicates that this is not decidable by the front end.

Example

```
a = rand(3,3);
b = rand(1,1);
mfe_size(b, 1:3, 1);
a * b;
```

1.4.25 E00025 – Error – Function diag is not defined for hypermatrices

The function diag is not defined for hypermatrices.

Example

```
diag(rand(3,3,3));
```

1.4.26 E00026 – Error – Function diag: Cannot determine if input is a vector

The function diag cannot determine at compile time if the parameter is a vector.

Example

```
a = rand(3,3);
mfe_size(a, 1:3, 1:3);
diag(a);
```

1.4.27 E00027 – Error – $funcname function: Expecting parameter $paramno to be constant

Some functions must have a constant parameter.

Example

```
a = 1;
b = rand(3,3);
size(b, a);
```

1.4.28 E00028 – Error – $funcname function: Invalid parameter $paramno

A parameter of a function is invalid.

Example

```
a = rand(3,3);
flip(a, (-4));
```

1.4.29 E00029 – Error – Lexer error: Invalid character '$char'

A invalid character is found in the input file.

Example

```
"Matlab does not allow double quoted strings"
```

1.4.30 E00030 – Error – Variable '$var' is already global.

A variable is defined as global multiple times.

Example

```
global a;
global a;
```

1.4.31 E00031 – Error – Global state of variable '$var' is ambiguous.

The global state of a variable on each variable usage must be decidable at compile time. See section 1.3 for more information about the propagation of type information at compile time.

Example

```
if (rand())
   global a;
end
a = 1;
```

1.4.32 N00032 – Note – Setting variable access to interval/constant size for variable: '$var'.

The `mfe_size` function allows to bypass the automatic size inference system and manually sets the size for a variable. The note indicates to which variable usage the manual size information are applied.

Example

```
mfe_size(a, 1:10, 1:10);
a = 1;
```

1.4.33 E00035 – Error – Different data types for variable $var ($dt1 vs. $dt2)

It is not possible to set the different data types to the same variable. Please use multiple variables.

Example

```
a = 1;
a = uint8(2);
```

1.4.34 E00036 – Error – break/continue statement only possible within a loop

Break and continue statement are only legal within a loop.

Example

```
break;
```

1.4.35 W00037 – Warning – Lexer warning: Use comma instead of space to separate expressions.

The Matlab and Scilab language is a space-sensitive language. The current implementation of the front end does not support space-sensitive parsing. If a space-sensitive expression is found, this warning is generated. Please rewrite your code and separate your expressions by comma instead of spaces.

Example

```
[2 3 4]
```

1.4.36 E00038 – Error – MFE size of variable '$var' is ambiguous.

The size state of a variable on each variable usage must be decidable at compile time. See section 1.3 for more information about the propagation of type information at compile time.

Example

```
a = 1;

if (a)
   mfe_size(a, 2, 2);
else
   mfe_size(a, 3, 3);
end

b = a;
```

1.4.37 E00039 – Error – Syntax: Unexpected end or $

The end (Matlab) or $ (Scilab) keyword is only allowed within an array access.

Example

```
a = (end);
```

1.4.38 N00040 – Note – MFE size of variable set here.

This note appears in combination with error E00038 and indicates where the size of a variable was set.

1.4.39 E00041 – Error – $funcname function: Expecting parameter $paramno to be a variable

Some functions require a parameter to be a variable.

Example

```
mfe_fixedsize(1);
```

1.4.40 E00042 – Error – $funcname function: Expecting parameter $paramno to be a string

Some functions require a parameter to be a string.

Example

```
mfe_pragma(1);
```

1.4.41 E00043 – Error – Number of output parameters does not match to function declaration

The number of output parameter does not match to the number of output parameter of a user-defined function.

Example

```
function [out1] = f()
   out1 = rand();
end

[out1, out2] = f();
```

1.4.42 E00044 – Error – Number of input parameters does not match to function declaration

The number of input parameter does not match to the number of input parameter of a user-defined function.

Example

```
function f(in1)
   disp(in1);
end

f(1, 2);
```

1.4.43 E00045 – Error – Binary operator $op: Cannot determine if left or right element could be scalar

It must be clear at compile time, if the right side or left side of a binary operator is a scalar. The error indicates that this is not decidable by the front end.

Example

```
a = rand(3,3);
b = rand(1,1);
mfe_size(b, 1:3, 1);
a + b;
```

1.4.44 N00046 – Note – Frontend is built in debug mode ... performance could be slow!

Internal note, should not appear is release code.

1.4.45 E00047 – Error – $funcname function: Expecting parameter $paramno to be a constant range

Some functions require a constant or constant range as parameter.

Example

```
mfe_read_image('test.png', 'this should be a constant range', 1:10);
```

1.4.46 E00048 – Error – Array access must match the number of dimensions of the variable.

The number of elements used for an array access must match the number of dimensions of the variable.

Example

```
a = zeros(3,3,3)
b = a(5,5)
```

1.4.47 W00049 – Warning – $funcname function: Function call is deprecated

The function or the function variant is deprecated. Please refer to section 1.2 to get a list of up to date functions.

1.4.48 W00050 – Warning – Constant propagation for operator '$operator' not implemented

Internal warning to indicate a missing feature.

1.4.49 W00051 – Warning – Condition couldn't be determined

Internal warning.

1.4.50 W00052 – Warning – Propagation of array access result not implemented

Internal warning.

1.4.51 W00053 – Warning – Variable $var not used.

Variable is defined but is never used.

Example

```
a = 1;
```

1.4.52 W00054 – Warning – Reading and writing the same variable in one statement could result in wrong behavior

Reading and writing the same variable in one statement could result in wrong behavior. See section 1.6.4 for more details.

Example

```
a = 1:5;
a(1,2:5) = a(1,1:4)   % Warning, produces wrong code

b = 1:5;
b(1,1:4) = b(1,2:5)   % Warning, produces correct code

c = ones(3,3);
c = c*c            % Warning, produces wrong code

c = 1:10;
c(1,[1,2]) = c(1,[2,1]) % Warning, produces wrong code
```

1.5 Limitations

1.5.1 Assigning different data types to one variable

It is not allowed to assign different data types to the one variable:

```
a = int32(1);
a = double(1);
```

Solution: Rewrite your code and insert a new variable.

1.5.2 Assigning different number of dimensions to one variable

It is not allowed to assign different number of dimensions to one variable:

```
a = ones(3,3);
a = ones(3,3,3);
```

Solution: Rewrite your code and insert a new variable.

Please note that in Matlab and Scilab a scalar is a two dimensional matrix of size $1x1$. For that reason, the following code is working:

```
a = 5;
a = ones(3,3);
```

1.6 Limitations (planned to be fixed)

1.6.1 Non-deterministic functions are duplicated

In this example, the rand function is duplicated into the for loop. As result, the for loop iterates not until the value a.

```
a = rand(1) * 10
t = 0
for j=1:a
   t = j
end
```

1.6.2 Support complex numbers in matlab

In Matlab mode the constant i and j is not supported. Matlab allows to override the i variable by other definitions. Supporting the i/j variable would require to add a more sophisticated analysis.

Constant number definition in the form `10i` is supported. Please rewrite the constant i and j by `1i`.

1.6.3 User defined function: Modifying a parameter inside a function affects the calling variable

```
function myfunc(in)
   in = 5
end

a = 10
myfunc(a);

a
```

1.6.4 Reading and writing the same variable in one statement may produce wrong results

The front end generates loop nests out of each Matlab or Scilab statement. Thereby, it does not check if one element overrides another element of an array. In that case, a temporary variable would be required.

In most cases, where no indexing or matrix multiplication is used, the correct result is generated. Otherwise, a temporary variable should be inserted by the end users.

1.6.4.1 Example 1 – Copying parts of an array

In the following example, parts of array a is copied to a. The problem is here that the indexes 2 : 6 are greater than 1 : 5.

```
a = 1:10;
a(1,2:10) = a(1,1:9)

=>    1    1    1    1    1    1    1    1    1    1
```

The resulting C code look like this

```
  for (v3 = 0; v3 < 9; ++v3) {
    a_data[(1 + v3)] = a_data[v3];
  }
```

and there the element 2 is overwritten in the first iteration. In the second iteration a wrong data from element 2 is read.

1.6.4.2 Example 2 – Matrix multiplication

```
a = rand(3,3);
a = a*a
```

Matrix multiplication requires 3 loops in the final C code.

```
  for (v7 = 0; v7 < 3; ++v7) {
    for (v6 = 0; v6 < 3; ++v6) {
      v8 = 0;

      for (v9 = 0; v9 < 3; ++v9) {
        v8 = (v8 + (a_data[v9][v6] * a_data[v7][v9]));
      }

      a_data[v7][v6] = v8;
    }
  }
```

As soon as one element in a is changed, the remaining algorithm works with the wrong data and produces wrong results.

1.6.4.3 Example 3 – Exchange

```
a = 1:10;
a(1,2:10) = a(1,1:9)

=>     2     2     3     4     5     6     7     8     9    10
```

Chapter 2

Usage

2.1 Command Line Parameters

Usage:

```
matrix-frontend [options] <input-file> <output-file>
```

Command	Info
--help	Show help message
--language <lang>	Specify input language (matlab, scilab, auto)
--print=<0|1>	Print result of statements
--printall=<0|1>	Print all variables at the end of each function
--simplify=<0|1>	Enable simplify pass (default 1)
--debug-parse=<0|1>	Enable debug for parser pass (default 0)
--debug-format=<0|1|2>	Enable debug for format pass (default 0)
--frontendlibpath	Output path to frontend library

2.1.1 Language

The frontend support Matlab and Scilab intput language. Per default, the input language is detected from the file name extension of the input file. .m files are Matlab and .sce/.sci files are Scilab. The input language can be manually set the –language command line option.

2.2 Test Directory

The installation of the Matrix Frontend includes a Test directory containing several test cases of Matlab/Scilab. The test cases are separated into subdirectories. Simple contains small test cases that tests specific parts and language features of the frontend. Apps contains application test cases.

The Test directory include one Makefile that can be used by the make command line utility. The Makefile includes a conviencient way to execute one or multiple test cases.

Within that directory you can run a test case by writing `make <test>.show` in the command line, whereby `<test>` represents the corresponding .m/.sce/.sci file. `.show` means here to pass the .m/.sci/.sce file through the frontend, generate a C file, compile the C file with gcc and execute the resulting application for generating the script output.

All makefile options are described in the follow table:

Command	**Info**
`make help`	Output makefile help
`make <test>.show`	Run frontend to general C file, compile the C file and execute application